ASSEMBLY LANGUAGE PROGRAMMING

FOR THE

TRS-80 MODEL 16

BY DAN KEEN & DAVE DISCHERT

ASSEMBLY LANGUAGE PROGRAMMING FOR THE TRS-80 MODEL 16™

.

ASSEMBLY LANGUAGE PROGRAMMING FOR THE TRS-80 MODEL 16 ™

BY DAN KEEN & DAVE DISCHERT

This book is dedicated to Alicia, Trisha, David, and Mark.

Contents

Getting Input—Testing for a Match Using the CMP Op Code—
The Branch Op Code

Acknowledgments

We wish to thank George M. Keen for his editorial suggestions and photographic contributions.

We also thank Kathryn Engle, marketing consultant with AT&T, for the many hours she spent editing our material.

We appreciate Mr. Steve M. Hluchanyk's technical assistance.

We especially thank Betty and Joe Tocci whose generosity made our research much easier.

We are very grateful to Radio Shack and to David Gunzel, Director of Technical Publications at Radio Shack, for supplying us with material for this book. They have graciously permitted us to use terms in the book which are found in the manuals supplied with the Model 16. By use of expressions such as "SVC block," the reader will not be unnecessarily confused by our inventing phrases that are not in keeping with his owner's manual.

Finally, we acknowledge Motorola and Mr. James T. Farrell III, manager, Technical Communications at Motorola, for their permission to reprint their technical information sheets on the MC68000 microprocessor.

The authors shall have no liability or responsibility to the purchaser with respect to any loss or damage caused either directly or indirectly by the use of this material.

Every possible effort has been made to ensure all information is true and accurate.

Introduction

One of the most powerful microcomputers currently on the market is the TRS-80 Model 16 manufactured by Radio Shack. Its great potential results from the use of an MC68000 microprocessor as the central processing unit.

Writing assembly-language programs to fully tap the resources of this computer is not a simple task. Indeed, assembly-language programming for any computer requires some study.

The documentation supplied with the Model 16 and its accompanying Assembler-16 manual contain much information, but the user must be an experienced assembly-language programmer in order to use it. While these manuals make no claim to be tutorial documents, many of the instructions and procedures are not clearly demonstrated and are difficult to understand.

This book is a supplement to the manuals supplied with the Model 16 microcomputer. It will enable you to begin writing assembly programs without your investing a great deal of time. We amplify many of the routines in these manuals and demonstrate how they can be put together to do various practical applications.

We do not describe each instruction in the MC68000 instruction set as there are other books available on the subject. This book, however, does address the specific needs of the Model 16 assembly-language programmer.

We develop complete subroutines that programmers can incorporate into their own programs. These include routines to clear
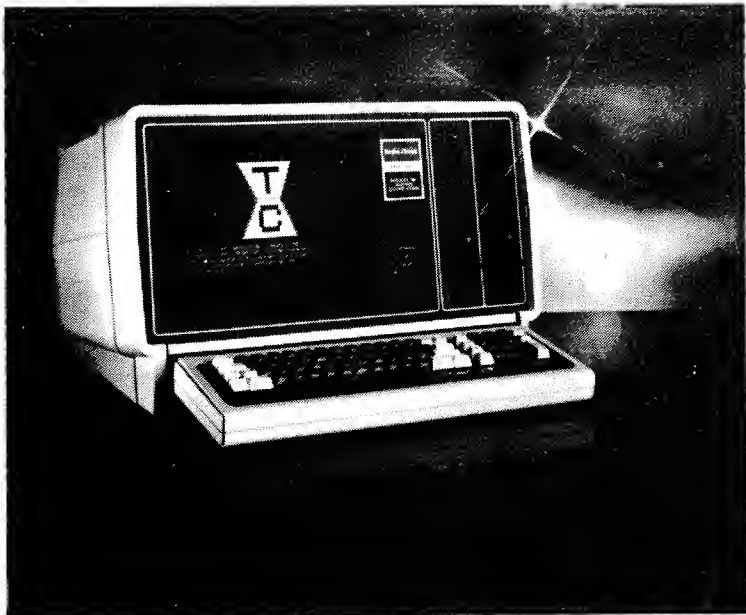
The TRS-80 Model 16 microcomputer. (Courtesy of Radio Shack)

the screen, print text on the video display, address the keyboard, output information to a printer, and create and maintain disk files. You may be a novice or an experienced programmer; however, each routine developed is thoroughly explained.

The material presented here will save Model 16 assembly-language programmers many frustrating hours that would otherwise be spent on trial-and-error attempts.

After completing this book, you may wish to delve deeper into the workings of the MC68000 and its instruction set. Our book will act as a springboard into more complex programming for this advanced microcomputer.

Radio Shack offers a Model 16 Enhancement option for their popular Model II and Model 12 computers. An extra circuit board converts the Model II into a Model 16, with the exception of a few features such as extra disk storage and a green video screen. All of the information contained in this book applies to the Model 16, modified Model II, and the modified Model 12. For simplicity, all references are for the Model 16; however they also include the Model 2 and Model 12.

Chapter 1
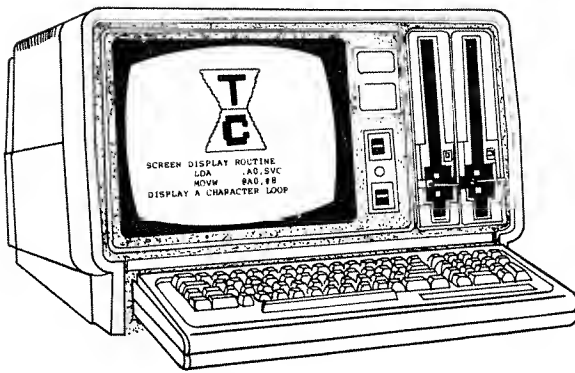
The Assembler-16 Package

The main advantages of writing programs in assembly language are speed and efficient use of memory space. High-level languages must go through an interpreter to decipher the commands and instructions of the language and put them into a machine-readable form. It is possible to write programs using binary or using the actual hexadecimal MC68000 instruction codes, but this would be needlessly laborious and time consuming.

An editor/assembler program allows us to write programs using letter symbols that make more sense to us as human beings. It converts those instructions into codes that are in the computer's mother tongue. An editor/assembler makes life easier for us in many other ways. For instance, it automatically computes address locations in memory for relative jumps (similar to GOTO statements in BASIC) by allowing us to use words and phrases to identify particular sections of a program.

DEVELOPMENT PROCEDURES

The Assembler-16 system actually consists of three separate programs. Using these programs, several steps are taken after you write your own program in order to get it into a set of instructions that the machine can execute.

First, the *editor program* is run. This allows you to write your program and create a "source" code. This program is saved and can later be loaded and easily changed. You cannot work directly on a

machine-language object code, which is the final form a program takes on to be executable. This is true when writing assembly programs for any type of microprocessor.

Next, the *assembler program* is run, which takes the source code and creates an intermediate object file. At this time the assembler will report to you any syntax or addressing-mode errors. If there are problems, you can correct the source code by going back to the editor program.

Once an error-free pass has been made, a program, called a *linker* must be run. The linker takes the assembled file of the program and develops a complete stand-alone machine-executable program, which it stores as a disk file.

Each of the three programs creates a separate file on the disk. The last file is the executable program, which is run from the disk operating system mode (TRSDOS 16 Ready). The source file is the code you work on if changes are necessary. The object file, is no longer needed.

Let us follow a complete sample development of a program. This will be the procedure to follow when you create your own assembly-language programs.

Programs are written with the editor. From the TRSDOS ready mode, type EDIT16. This loads the editor program.

Let's assume we have written a mailing list source code, by following the instructions in the Assembler-16 manual. This source listing must be stored on the disk. A logical file name for our sample session might be MAILLIST. The extension we must give our file name is /SRC to tell the computer that this is a source listing. In the command mode of the editor program, type SAVE MAILLIST/ SRC.

Next we exit the editor program and return to the disk operating system ready mode. To assemble our program, type ASM16 MAILLIST. This will load the assembler and create an intermediate file called MAILLIST/OBJ, representing an object file. Notice that it was not necessary to use the extension /SRC when we told the assembler the name of the program to assemble.

For some strange reason, the writers of the Assembler-16 make us create a so-called *control file* to direct the linker program. It seems to be an unnecessary step, one that could perhaps have been incorporated into the assembler itself. In any event, a discussion of the complete procedure follows.

Type EDIT16 and reload the editor program. When prompted with C?, type IN to insert the following two lines:

2

```
INCLUDE     MAILLIST
END
```

We suggest that INCLUDE and END be placed in the operand column (as opposed to the label column), so use the tab key on the computer before entering them (label and operand columns of the assembler are discussed in the next chapter). The file name may also be placed in the next tab position. Therefore the first line could be entered by hitting the tab key, typing INCLUDE and hitting the tab key again and typing MAILLIST.

To exit the insert mode, type the symbol ! and the C? will reappear. Save this on disk with the file name MAILLIST/CTL. The extension /CTL indicates to the computer that this is the control file. Enter the word QUIT and again return to the TRSDOS 16 ready mode.

Finally, enter LINK16 MAILLIST and the machine-language program will be placed in its completed form.

The program is executed by entering MAILLIST from the TRSDOS 16 ready prompt.

Chapter 2


```
SCREEN DISPLAY ROUTINE
        LDA      A0,SVC
        MOVM     #A0,#B
DISPLAY A CHARACTER LOOP
```
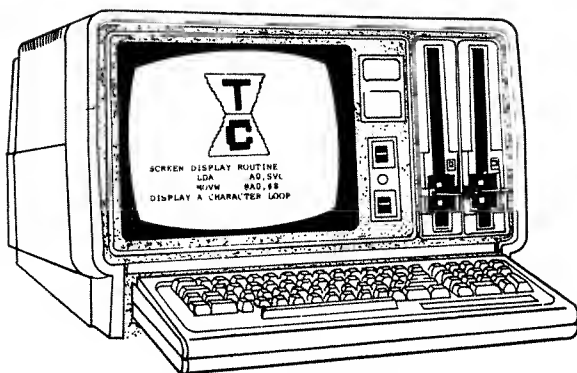
Developing a Source Code

In this chapter we will talk a little about the MC68000 micro-processor (Fig. 2-1) and how we write instructions for it using the Radio Shack EDIT16 editor program.

OP CODES

When you first learned to program in a high-level language, it was necessary for you to learn the various reserved words and symbols. In BASIC, for example, you would have had to learn that the PRINT command instructs the computer to display values or text on the video screen. In time the commands of any language become second nature to you.

In the same way, assembly language consists of groups of letters that symbolically represent instructions for the computer. The big difference is that now we must work directly with the microprocessor.

Many operations, such as arithmetic calculation, are harder to do in assembly language than in high-level languages, and some are easier such as moving blocks of information from one area in memory to another.

The reserved words or commands are called *op codes*, which is short for operational codes. These are the actual instructions that the MC68000 microprocessor chip understands. To help us remember what the various op codes do, their names often resemble

4
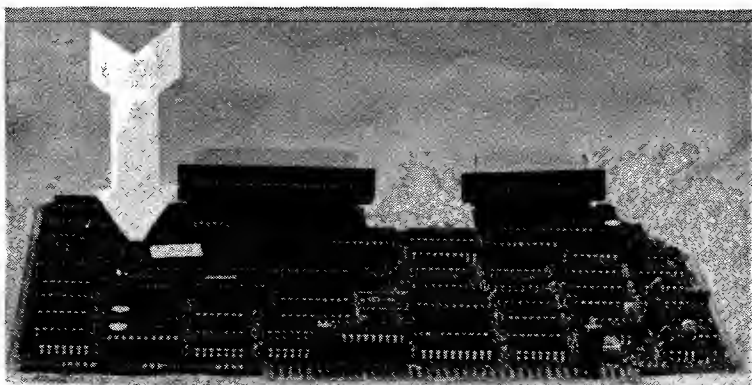
Fig. 2-1. The MC68000 microprocessor. The arrow points to the microprocessor as it sits among associated Model 16 circuitry. (Photograph by George M. Keen)

their functions. For example, LD is the op code for loading a register with a value.

There are about 20 op codes which we can use in our programs that the microprocessor does not understand. These are called *pseudo op* codes. Although these instructions are not understood by the MC68000, they have meaning to the editor/assembler program. The assembler translates these terms into machine code for us.

By using pseudo ops, we can write programs faster and with greater ease. We will be using pseudo ops in the routines developed in later chapters.

REGISTERS

Inside the central processing circuit itself are small areas of RAM (random access memory). These spaces are called *registers*. Their purpose is not to hold cash, but they do hold values. The MC68000 has many registers. The general-purpose registers that we will be dealing with in this book fall into two categories: address and data registers.

Each register can store a maximum of four bytes of information. Think of registers in the central processing unit as temporary storage areas. Data is placed in registers to be manipulated, moved, or compared. Some registers must have certain values placed in them before executing some of the op codes.

LOADING REGISTERS

Loading a register means taking data from one register or

5

memory address and copying it into another register or memory location. The source address, the register where the data was copied from, remains unchanged. Therefore, an LD command does not alter the contents of the "from" address, only the destination address.

Since registers in the MC68000 can contain instructions up to four bytes long, we must tell the computer how many bytes a specific instruction will act on. The letter B represents one byte; W represents two bytes, which is referred to as a word; and L represents a long word, that is, four bytes.

By following the load instruction with one of these letters, we can tell the computer how much room we will require in the register. For example, the instruction to load the A0 register with the decimal number 10 would look like this:

$$\text{LDW} \qquad \text{.A0, \#10}$$

The suffix W attached to the LD command causes the register to utilize two of its bytes to store the number 10. The value on the right-hand side of the comma is loaded into the register whose name appears on the left-hand side of the comma. You will note the use of a *pound sign* (#) placed before the number 10. This tells the assembler to load the information immediately following the pound sign. The function of the period before the register name is covered under Direct and Indirect Addressing, which follows.

In this example the number 10 can be stored in one byte. Therefore the instruction could have used LDB instead of LDW. However, because a "word" was specified, the assembler will fill the upper byte with a zero, and will store 10 in the lower byte of the register. Whether you use LDB or LDW depends on what you want to do with that register at a later point. Some commands only allow operations to be performed on a word of information and not a byte. If, at a later date, we needed to access a number stored in a register by using an op code that requires a word, we are sure that there would not be any random value in the upper byte of the register as long as a 10 was loaded with the LDW instruction.

DIRECT AND INDIRECT ADDRESSING

A *period* placed before a register signifies a direct operation on that register. Again in the instruction:

$$\text{LDW} \qquad \text{.A0, \#10}$$

note the period placed in front of A0. This means that the value 10 is placed directly in register A0.

It is possible to place the value 10 or any other value we may choose into an address in the computer's memory. This can be accomplished by first loading the desired memory address into a register, A0 in this case. This time we do not want to place the value directly into register A0, but rather into the memory location "pointed to" by the address stored in A0. In other words, we want to indirectly address A0. The @ symbol instructs the assembler to do that very thing, as in

<p align="center">MOVW   @A0, #10</p>

The number 10 will be stored in the memory location specified by register A0. The MOVe command is used to load a register indirectly. It performs the same function as the load (LD) op code; however, the LD instruction is not capable of loading a register indirectly. Later in this chapter we will discuss the various op codes needed to load registers in different addressing modes.

THE EDIT16 FORMAT

The editor program acts on the instructions we enter by determining the function of the op code and the position of items we place in various columns on the screen. The column titles are

<p align="center">LABEL   OP CODE   OPERAND   COMMENTS</p>

Labels are words or phrases used to identify particular sections of a program. It follows then that we can use a label as a location to jump or branch to from other areas in the program. These are similar to pseudo ops in that they are not a part of the MC68000 instruction code. The assembler computes the actual address location of a label. When we use a label in the operand column, the assembler will instruct the computer to use the address. For example, we could tell the computer to branch or jump to a label name and it would jump ahead or behind by the correct number of bytes. The next instruction to be executed would then be the address indicated by the label.

We may also wish to load a register with the address of a particular label. Assume we have set up the phrase SVC block (a type of buffer area, which is discussed in Chapter 3) in the label column. In order to load the address where the SVC block begins

into register A0, we could write the instruction

LDA   .A0,SVC BLOCK

(LDA is an op code which we will cover in a moment.)

The next column in the editor is where the op code is placed. This is the actual "mnemonic" or instruction.

There are no hard and fast column positions, but there must be at least two spaces between the items placed in each column.

Next is the *operand column*. Not all instructions have an operand; some have two. The source and destination addresses or registers are specified in this column. A comma separates operands if there is more than one part.

In most cases, the source will be identified on the right side of a comma, and the destination on the left, but there are exceptions. The value on the right is loaded into the registor or memory location on the left. This is true of all load and move instructions. In

LDW   .A1,#13

the decimal number 13 is loaded directly into register A1.

The exception to the rule is the ST or STore op code. It takes the value to the left of a comma and stores it in the address or register indicated to the right. In

STW   .A2,@A0

the value currently residing in register A2 is loaded into the memory location whose address is stored in A0. Note that the positioning of the source and destination operands are in contrast to the load and move instructions; they are in the opposite format. But in both cases, the source address will still contain the same value after the instruction has been executed. Only the destination register will be altered.

Finally, the *comment column* is located at the far right. This is where the programmer can place comments or remarks to help document his program. The assembler ignores these comments when it creates the object code, but they remain in the source listing.

Comments do not necessarily have to be placed in the comment area. By preceding any comment with an asterisk (*), remarks may be added. This is useful to help identify certain lines in a program

since they can be sectioned off by using a whole row of asterisks or other symbols. It is advisable to place the asterisk and begin a comment line either in the label or comment column. Avoid starting a comment in the operand column, as assembler errors sometimes result.

ADDRESSING MODES

Addressing mode refers to the way in which a register can be accessed or "talked" to. For example, the instruction

> LDW   @A0, #264

is illegal. The MC68000 does not understand that instruction. It makes sense to us, though. We may try to use that format to load the decimal number 264 into the memory address location stored in register A0. To perform this task we must use the MOV op code:

> MOVW   @A0, #264

This form is acceptable to the microprocessor.

So, how do we know which addressing modes are legal and which are illegal? This is a difficulty that awaits every beginning assembly-language programmer regardless of the type of micro-pressor being used. Unfortunately, there are few hard and fast rules here. It is kind of like learning the spelling rules of the English language in that there are many exceptions to the rules. In school we were taught "i before e except after c." Just when we thought we knew that rule, we came up against such words as "reign," "neither," and "their."

Our best advice is that if you cannot do something one way, try another even if you must use an indirect method. There is bound to be another way to accomplish the same thing. It may not always be the easiest way but so long as the end result is the same, you are in business. The Assembler-16 manual also shows valid addressing modes under each op code heading which can assist you finding the correct format for a particular operation you may need to perform.

A good example of accomplishing a task indirectly can be found in the error handler routine in Chapter 9. In that program, it is necessary to take a two-byte long number, which is located at byte-offset two in a block of memory pointed to by register A0, and move that number into byte-offset six of the same block. To put all that in English, what we are trying to do is to load one byte-offset

register (discussed at the end of this chapter and in Chapter 3) into another byte-offset. Such a thing is not directly possible, so we need to do the job indirectly. It will require two steps.

```
LDW    .A1,2@A0    *LOAD THE NUMBER STORED IN PO-
                   *   SITION 2 OF THE ADDRESSING
                   *   POINTED TO BY A0 INTO A1
STW    .A1,6@A0    *LOAD (STORE) THAT NUMBER
                   *   BACK   INTO   THE   ADDRESS
                   *   REFERENCED BY A0 BUT AT
                   *   BYTE-OFFSET 6
```

So, those lines of programming will accomplish the same goal.

Next we will present a few of the most commonly used addressing mode formats. These formats are the building blocks of many routines.

As a general rule, the load, move, and store op codes (LD, MOV, ST) are used to address registers in one way or another. The ways used to access registers are direct, indirect, and indirect with a "byte-offset."

DIRECT ADDRESSING

In cases where a register is to be loaded directly with a value, the LD instruction may be used. The letters "B," "W," or "L" should accompany the op code to tell the microprocessor the size of the data to be operated on, namely a byte, word, or long word. The instruction

```
LDW     .A1,#H'FF
```

loads register A1 with the hexadecimal number FF. The instruction

```
LDW     .A1,#13
```

loads register A1 with the decimal number 13, and

```
LDA     .A0,MESSAGE
```

loads register A0 with the address of the label MESSAGE. You may be wondering where the suffix "A" came from as we have previously stated that the suffixes are B, W, and L. It so happens that LDA is a complete MC68000 instruction. All four bytes of the register are

affected, and it is not necessary to tag a data size character onto the end. So, LDA is an instruction that means load an address that immediately follows into the specified register. Note that the pound sign (#) is not necessary in front of MESSAGE when using this instruction.

INDIRECT ADDRESSING

Indirect addressing refers to the loading or retrieving of information from an address pointed to by a register. For example:

$$\text{MOVW} \qquad @A0, \#264$$

loads the memory location whose address is stored in A0 with the decimal number 264. The instruction

$$\text{MOVW} \qquad @A0, \#VALUE$$

loads the address pointed to by A0 with the number stored in the location referred to as VALUE. This label is set equal to a number elsewhere in the program.

INDIRECT ADDRESSING USING BYTE-OFFSET

In the next chapter, we will discuss the concept of creating an area of memory space within a program to be used when calling routines that are built into the disk operating system. This area is commonly called a *supervisor block* or *SVC block*. The SVC block must be made up of 32 consecutive bytes in memory. It is necessary to place certain values inside this area before jumping to one of the system routines. This is done by indirectly addressing a register using a *byte-offset*, explained in Chapter 3. The move instruction can be used when doing this type of addressing. For example, the line

$$\text{MOVW} \qquad 8@A0, \#1$$

stores the decimal number 1 into the seventh and eighth byte of the block whose address is indicated by register A0. This is referred to as byte-offset eight and nine. To see how byte-offset is calculated, examine the supervisor block setups in the appendices.

$$\text{STW} \qquad .A2, 8@A0$$

stores the number currently in register A2 into the seventh and

eighth byte of the block whose address is indicated by register A0. Note that this is backwards from the MOVe and LoaD codes where the source contents are on the right of the comma in the operand and the destination is on the left. The STore instruction performs the same function as MOVe in this case. But MOVe does not have an addressing mode to load a value directly from an address register into a register indirect with byte-offset. That is the reason for choosing the STore op code.

These examples should prove to be a handy reference as you begin assembly-language programming on the Model 16 computer.

Chapter 3
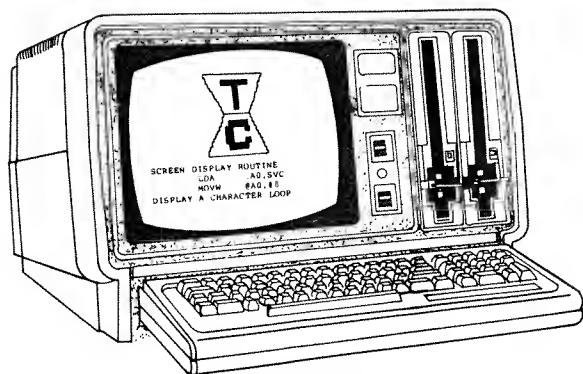
Supervisor Routines

The disk operating system (referred to simply as the "operating system") of the computer consists of a series of machine-language routines that the microprocessor uses to communicate with the keyboard, video display, disk drives, modems, and printers. Some of the routines are available for us to call on to do work. These routines are referred to as *supervisor calls*, or SVCs for short.

We can save a lot of time writing assembly programs by utilizing supervisor routines. These can help us print messages on the screen, direct output to a printer, get information from the keyboard, and handle disk input/output operations. Programming would be a very tedious job if we had to write our own routines to perform these various tasks by giving instructions directly to the disk controller and other circuitry.

By properly setting up the MC68000s registers and establishing a buffer area in RAM called an SVC block, we can choose one of the many supervisor routines and execute it from our own machine-language programs.

THE SVC BLOCK

Before calling a supervisor routine, an area of RAM (random access memory) must be selected to act as a buffer. This buffer area is usually referred to as the SVC block. This is in keeping with your owner's manual.

Each supervisor call has a number by which the operating

system identifies it. This is known as a *supervisor function code*, or an *SVC number*. Every SVC block must have this indentifying number placed in the first word (first two bytes) of the block before the routine is called.

The supervisor number and other parameters are placed within the block to initialize it prior to execution of the routine. Also, values can be passed back and forth between a program and the supervisor routine via this buffer.

An SVC block of space can be placed right within the assembly-language program by using a label and instructing the assembler to set aside 32 contiguous bytes of memory. The assembly listing should be set up similar to the following:

```
SVC BLOCK
          RDATAB     32,0
```

It is customary to use SVC BLOCK as the identifying label. RDATAB is a pseudo op, which along with the operand 32,0 sets aside a series of 32 consecutive bytes in memory and fills them with zeros.

The block must contain 32 bytes, which is the equivalent of 16 words. To address a particular word within the block we use a number to indicate the starting byte's position. This is called the *byte-offset* number.

The first byte in a block has a byte-offset number of zero, the second byte position is indicated by a one, the third by a two, and so on. As an example, let us assume register A0 has been loaded with the address location of the beginning of the SVC block, and that a number has been placed into Al. The instruction:

```
          STW       .Al,6@A0
```

will cause the value stored in Al to be placed in the seventh and eighth bytes of the block. This is an offset of six bytes, which is actually the seventh byte beyond the beginning address pointed to by register A0. Figure 3-1 shows the relationship of the actual byte positions and the so-called "byte-offset" position.

Since a word is two bytes long, the byte-offset number must always be an even number. The first three words in a supervisor block must always store the same type of information, regardless of the routine to be called. These include the identifying supervisor number and an error code placed in the block by the operating
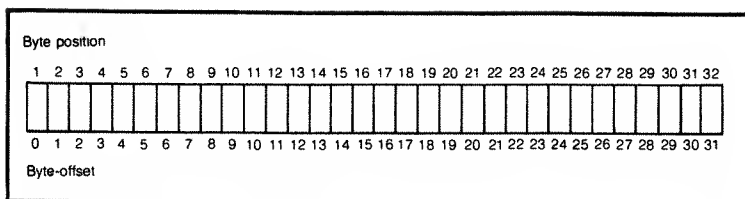
Fig. 3-1. Demonstrates the relationship between "byte-offset" and the actual byte position within the block of memory in a program, which has been set aside as a buffer to call a supervisor routine contained in the disk operating system.

system if an error occurs during the execution of the routine. Also, byte-offset four and five must always contain a zero.

Some routines, such as those that control disk input and output, will also require certain parameters to be placed in other positions within this buffer.

To set up a supervisor call we first establish an **SVC** block. Then we must place the identifying disk operating code and any other parameters particular to that specific call into the block in the appropriate positions. Execution of the routine is initiated by the instruction BRK   #0.

A ROUTINE TO RETURN TO DOS

The first routine we shall develop will cause the computer to exit a machine-language program and jump to the TRSDOS 16 ready mode. Usually you will use this routine at the end of a program to terminate its execution.

Figure 3-2 shows the complete "return to DOS ready" routine. This can be entered and assembled exactly as it appears. Nothing more needs to be added.

If you have dabbled in assembly-language programming for one type of microprocessor or another, you will notice that this program does not contain an ORG pseudo op. The ORG op code is usually

```
START        LDA        .A0,SVC BLOCK
             MOVW       @A0,#264
             BRK        #0
SVC BLOCK
             RDATAB     32,0
             END        START
```

Fig. 3-2. A complete jump to the disk operating system routine.

15

used to establish the starting address of a machine-language program. While Radio Shack's assembler does recognize the ORG statement, it is not necessary to place it in the program. If no origin is specified, the assembler will simply load the machine-language program into the first available space in RAM once the disk operating system and any utility programs have been loaded.

The first instruction

<div align="center">

LDA   .A0,SVC BLOCK

</div>

loads the beginning address of the area whose label identifies it as SVC BLOCK into register A0. Next,

<div align="center">

MOVW   @ A0,#264

</div>

moves the number 264 into the first word of the SVC block area. Two bytes are used because of the W attached to the MOV command. As we mentioned in the last chapter, the @ symbol ("at" sign) indicates an indirect load where the register itself is not loaded with the number but rather the address that the value in A0 "points to."

To actually execute the supervisor call we give the command BRK #0 which causes the computer to jump to the address in memory where the supervisor routine is located. It also internally stores away the address of the next instruction to be carried out upon completion of the supervisor call. This is somewhat similar to the commands GOSUB and RETURN in the BASIC language.

Every assembly-language program must be terminated with an END statement. But prior to that we must create our supervisor buffer, hence the instruction

SVC BLOCK
          RDATAB   32,0

It is important to place the op code and the operand on a different line in this case. This makes the label universal in that this instruction can be referenced from anywhere in the program.

We should also point out that the reason for filling this block with zeros is two-fold. First, this makes sure that there are no stray characters or control codes present within the block; we are starting with a clean slate so to speak. Second, all of the supervisor routines must have zeros placed in byte-offsets four and five, as

```
START          LDA         .A0,SVC BLOCK
               MOVW        @A0,#JUMPDOS SVC NUMBER
               BRK         #0
SVC BLOCK
               RDATAB      32,0
JUMPDOS SVC NUMBER
               EQUW        264
               END         START
```

Fig. 3-3. An enhanced version of a jump to the disk operating system.

these bytes are reserved for the disk operating system.

Figure 3-3 lists the identical routine. However, here we have placed it in a form that makes it more universal. By that we mean the various parameters such as the SVC block and the SVC number have been set up as areas in RAM which we can easily change in order to execute other supervisor calls. The pseudo op EQUW, which stands for EQUate a word, causes the assembler to replace every occurrence in the program of JUMPDOS SVC NUMBER with the indicated value, namely decimal 264. The label takes on the value of the operand. It does not take up any space in memory when the source code is assembled. This is easily seen by examining a printout of an assembled listing.

The assembler printout is in the format of columns, each representing a different parameter. A sample section may look like this:

```
25    000062    4D454E55    25   MENU    TEXT    'MENU'
26              0000000D                 EQUW    13
27    000066    3D3D3D3D                 TEXT    '===='
```

The first column shows the line number of the source code. Here we see lines 25, 26, and 27. Next is the relative memory address. This value is displayed in hexadecimal form. The first line of the program has a relative address of zero. The op code TEXT (described later in this book) is shown beginning on relative address 62. There are four letters in the word "MENU," which should push the relative address of the next instruction to 000066. You will notice that the next instruction is EQUW 13 and it does *not* take up any room in memory. The op code TEXT, which sets aside space and fills it with equal signs, is the next instruction which is given memory space. Keep this definition in mind. You cannot use EQU to hold values in a particular area of memory as you would need to do to

place a terminator character (as shown in Chapter 5).

The column after the relative address is the actual generated instruction code in hexadecimal notation. The column to the right of that indicates the line number in the source listing where that particular label was defined.

The last columns are the same as the editor program, namely the label, op code, operand, and comment columns.

Now, getting back to our explanation of the jump to DOS routine, we come upon the label START. It is located at the beginning of the program and it identifies the first instruction to be executed. While START is used here, BEGIN or any other valid label could be employed to represent this address. This label is used as an operand for the END command, telling the computer the memory address it is to begin executing once the program has been loaded into memory. The first line of a program may not necessarily be the point at which program execution is to commence. Assembly-language programmers will frequently place tables or equate values to labels at the top of their programs. To prevent the computer from trying to execute these lines, a label such as START is placed at the first instruction past them.

Both routines listed in Figs. 3-2 and 3-3 are complete and ready to run as they stand.

Chapter 4

SCREEN DISPLAY ROUTINE
LDA   A0,SVC
MOVM   @A0,88
DISPLAY A CHARACTER LOOP
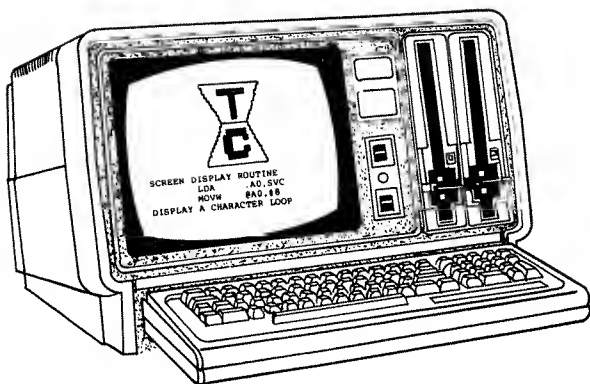
Screen Formatting

Whatever language you use or whatever type of program you write, one of the very first tasks you will probably ask your program to do is to clear the video display. By clearing the screen and starting with a "clean slate," you pave the way for printing text and instructions to the user.

Once the video display is cleared, it is usually desirable to place the cursor in the upper left-hand corner of the screen. This is referred to as the cursor's "home" position. The cursor can also be moved to any position on the screen with the position cursor routine described in Chapter 5.

The Model 16 allows several styles of printed characters on the screen. There are two sizes of letters as well as selections between normal and inverse video characters.

In the small size, 80 characters can fit horizontally across the screen on one line. Usually, this is the preferred size and is therefore considered the "normal" mode. A 40-character mode doubles the width of each letter and consequently only half as many characters can fit on a line.

In its normal format, this computer displays green letters on a black background. Through software control we can cause the background to be green and the letters to be black. This reversed-character mode of printing is known as *inverse video*. On a Model II computer that has been modified to include the MC68000 circuitry, the inverse video mode is black letters on white, since the Model II

does not have a green phosphor picture tube.

There are two different routines we can use to do the job. The following subroutine listing clears the screen, homes the cursor, prepares the normal/inverse mode for normal green on black printing, and sets the letter size to 80 characters per line.

```
      LDA       .A0,SVC BLOCK   *LOAD ADDRESS OF BLOCK INTO A0
      MOVW      @A0,#7          *LOAD BLOCK WITH SVC NUMBER
      MOVW      6@A0,#1         *LOAD BLOCK-SET 80 CHAR. MODE
      MOVW      8@A0,#1         *LOAD BLOCK-SET NORMAL VIDEO
      BRK       #0              *JUMP TO SUPERVISOR ROUTINE
SVC BLOCK                                                      .
      RDATAB    32,0            *ESTABLISH BLOCK-FILL WITH ZEROS
```

In its present form the routine is not a complete stand-alone program. We will discuss why it cannot be run without the addition of a few more instructions in just a moment.

The supervisor number, which the disk operating system requires to identify the routine, is seven. This number is placed into the supervisor block with the instruction:

MOVW   @A0,#7

Byte-offset six in the SVC block establishes the size of the letters; as in

MOVW   6@A0,#1

A numerical value of one places the video in the normal mode, which is 80 characters per line. The 40-character mode is kicked in by loading a zero in this position.

The normal or inverse video format is determined by the value in the eighth byte-offset, as in

MOVW   8@A0,#1

A value of one is normal, while a zero places the display in the reverse mode.

The asterisks in the listing allow us to make comments or remarks and store them within the source code listing for our own use. This extra documentation is not assembled and does not become a part of the final object code. So feel free to liberally place as many comments within a program as you need. Comments do not hamper the speed of program execution nor do they take up any

20

memory in the final object code form.

Now back to the reason why this program cannot be executed in its present form.

First, the routine needs an identifying label at the beginning and at its end. As was discussed in Chapter 3, a label placed at the beginning must also be placed in the operand column of the END pseudo op. The Assembler-16 manual shows the operand for END as being optional, but we ran into trouble when we left it off, and so therefore recommend using one. The label should point to the location in the program where execution is to begin. This is usually the top of the program, but if the first few lines of a program are used to equate, store, or otherwise define information, execution must begin after this area.

The second problem that would occur from directly running this program is the failure to return to the disk operating system ready mode, or to at least jump around the SVC block. Once the computer executed the subroutine (by way of the BRK   #0 line), it would attempt to carry out the codes generated by creating the SVC block. The machine would try to interpret these codes as instructions and would therefore either do something unpredictable; or, if it was an illegal code, stop and report an error message on the screen.

A fully complete clear screen and return to TRSDOS 16 routine follows. It uses supervisor call number eight which we will discuss in a moment.

```
START       LDA       .A0,SVC BLOCK
            MOVW      @A0,#8
            MOVW      6@A0,#3Ø
            BRK       #0
* JUMP BACK TO DOS NEXT
            MOVW      @A0,#264
            BRK       #0
SVC BLOCK
            RDATAB    32,0
            END       START
```

Notice that the routine above calls more than one supervisor routine, yet it only has one SVC block. Since only one supervisor routine is called at a time, there is no reason why one block of dedicated memory space cannot be shared by all of the supervisor routines in the program.

ANOTHER METHOD

The Model 16s disk operating system also has a built-in routine which allows us to output a character or a control code to the video

display. A list of all valid control codes is given in the owner's manual supplied with the computer. From that chart, we find that a hexadecimal 1E, or a decimal equivalent of 30, will cause the video display to be blanked out. The character or code that we want must be placed in byte-offset six of the supervisor block. Either the hexadecimal or corresponding decimal number can be used. The assembler will interpret either one. The letter H followed by an apostrophe and the hexadecimal number is the convention to use for assigning a hexadecimal value.

```
        MOVW      6@A0,#30        *using  a  decimal

                            or

        MOVW      6@A0,#H'1E      *using hexidecimal
```

The supervisor number for this call is eight. It is a very simple setup procedure, as follows:

```
          LDA      .A0,SVC BLOCK
          MOVW     @A0,#8
          MOVW     6@A0,#30
          BRK      #0
SVC BLOCK
          RDATAB   32,0
```

Chapter 5
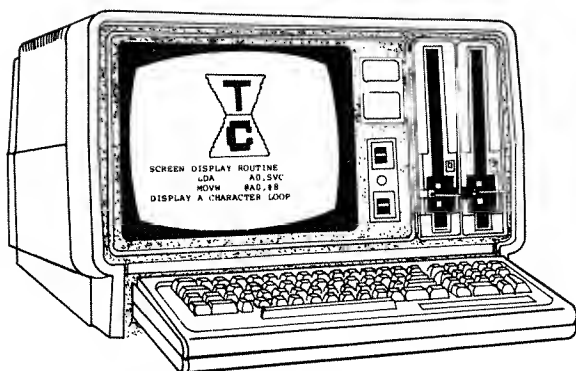
Displaying Text on the Screen

No matter what the application of a program may be, it will almost certainly require the display of information on the video tube. The computer can communicate with the operator via the screen by displaying information such as the results of a computation, instructions for using a program, or perhaps a *menu*, that is, a list of options from which the user may make a selection.

In this chapter we will tackle the job of displaying lines of text on the screen.

The disk operating system gives us several calls addressing the video display. One routine outputs a single character or a control code to the screen. This "display a character" call, as we shall refer to it, has a supervisor number of eight.

The second routine allows us to display an entire line of codes and text (a string of characters, words, and phrases). The supervisor number nine identifies this "display a line" routine.

Why, you may ask, are two different routines employed? Well, each has its own merits. While the display a character call is capable of only printing one character or sending one control code to the screen at a time, it has an advantage in that any ASCII code from 0 through 255 can be used. This means that it can be used to display the various graphic characters which are contained in the Model 16s video character generator. The ASCII code for these shapes begins with 128.

Many times it is necessary to place words, phrases, and sen-

tences on the screen. For this job we may resort to the supervisor routine which will output an entire line of text to the display. However, the drawbacks to this call is that it is limited to characters and control numbers whose ASCII code falls between 1 and 127. Control codes extend from 0 to 31. Numbers, symbols, and upper- and lowercase letters have ASCII values between 32 and 127. Therefore, no graphics can be displayed by using this method.

Before we go any further, there is one call that we should discuss.

POSITIONING THE CURSOR

Prior to calling any routine to display data on the screen, you may wish to place the cursor at the screen location where you desire printing to begin. Both the display a character and the display a line routine start printing at the cursor's current resting spot.

This is a fairly straightforward routine with the supervisor number being placed within the SVC block at the usual address of byte-offset zero, the row position in offset six and seven, and the column position in eight and nine.

The row position refers to the Y-axis, that is, up and down or vertical placement. This can be a number from 1 through 24, since the video display is set to have a maximum of 24 horizontal lines.

By the same token, the column refers to the x-coordinate, which can be 1 through 80.

The following listing moves the cursor to the center of the video display. We will locate the cursor 12 lines down from the top of the screen, roughly in the middle of the tube. Since there are 80 character positions across a horizontal line, we chose 40 as the x-coordinate.

```
          LDA      .A0,SVC BLOCK
          MOVW     @A0,#10    *IDENTIFYING CODE
          MOVW     6@A0,#12   *LINES DOWN FROM TOP OF SCREEN
          MOVW     8@A0,#40   *POSITIONS FROM LEFT SIDE OF TUBE
          BRK      #0
SVC BLOCK
          RDATAB   32,0
```

DISPLAYING A LINE OF TEXT

Once we have placed the cursor at the desired location, we are ready to output text to the screen.

In conjunction with the display a line supervisor call, we need to define a block of text. Radio Shack's Assembler-16 program contains a directive (a pseudo op) that eases this task. The instruc-

tion is TEXT. With it, we can set aside a block of RAM within our program and in which we can store strings of characters. By using a label to define this section, we can access it from various points within a program.

In the sample listing below, the computer is instructed to display the statement PREPARE PRINTER FOR OUTPUT.

```
          LDA       .A0,SVC BLOCK
          MOVW      @A0,#9
          MOVW      6@A0,#26
          MOVW      8@A0,#13
          MOVL      10@A0,#MESSAGE
          BRK       #0
SVC BLOCK
          RDATAB    32,0
MESSAGE   TEXT      'PREPARE PRINTER FOR OUTPUT'
```

Let's thoroughly examine this listing. First, the address of the SVC block is loaded into register A0.

The identifying supervisor number for this routine is nine and is loaded into the SVC block in the first and second bytes of it.

Next the computer needs to know several things about our line of text that we want to print. We must tell it how many characters make up our message. This information is placed in the "block" at byte-offset six. In byte-offset eight we place what the owner's manual calls the "terminator character." The term "terminator" gives the impression that this ASCII code marks the end of the message. This is not true. The machine does not need to have a terminator since the number we place in byte-offset six representing the length of the text to be displayed, is in control. The code placed in offset eight is the character which we want to send to the screen after our message has been displayed. The thinking here is that typically a programmer would want to send an ASCII 13 at the end of a message to give a line feed and a carriage return. Here you will use a decimal 13 to create a carriage return.

The address of the first byte of our message must be put in byte-offset ten. In our sample listing, you will note that we defined this address by referring to the text we wish to display with a label, namely MESSAGE. An important note must be made. Notice how we have always MOVed a "word" but when we deal with a memory address or a label which represents an address, we must use a "long word." It takes four bytes to store the value of an address and should you mistakenly define the above example as MOVW instead of MOVL, no error would be generated and your program would run, but nothing would happen on the screen.

As we have pointed out in Chapter 4, this routine would need an additional call to return execution back to the TRSDOS ready

mode. This would have to be carried out before the computer would be allowed to reach the SVC block.

DISPLAYING A CHARACTER

Often it is desirable to direct a single character or control code to the screen. Control codes are mainly used to perform such jobs as maneuvering the cursor and initializing the video display. You may remember this supervisor call from Chapter 4 where we cleared the screen by sending an ASCII code of 30 to the routine.

The ASCII code for the character to be sent to the screen is placed in byte-offset six within the SVC block. A list of available control codes and their functions is given in the Assembler-16 manual and therefore need not be shown here.

The ASCII value of 13 (0D in hexadecimal) causes a line feed, which refers to moving the cursor down to the beginning of the next line. We will use this code in our sample routine, as follows:

```
          LDA     .A0,SVC BLOCK
          MOVW    @A0,#8
          MOVW    6@A0,#13      *CODE TO BE OUPUT TO SCREEN
          BRK     #0
SVC BLOCK
          RDATAB  32,0
```

PUTTING SEVERAL ROUTINES TOGETHER

Now that we have discussed all of these short routines, let's put them together in one program to do something useful.

In the following complete listing, we will instruct the machine to clear the screen, position the cursor, display a message, and return to the DOS ready mode. The cursor is positioned so that the message will be horizontally centered on the top line of the screen.

All of the calls can use the same defined SVC block. As a matter of fact, once we load register A0 with the address of the block, there is no reason to do it again at the beginning of each routine. At no time do we use that register anywhere else in the program, so its contents remain intact.

```
START
CLEAR SCREEN
          LDA     .A0,SVC BLOCK
          MOVW    @A0,#8          *FUNCTION CODE
          MOVW    6@A0,#H'1B      *ASCII CODE TO CLS
          BRK     #0
POSITION CURSOR
          MOVW    @A0,#10         *FUNCTION CODE
          MOVW    6@A0,#1         *ROW POSITION (TOP/BOTTOM)
          MOVW    8@A0,#31        *COLUMN POS. (LEFT/RIGHT)
          BRK     #0
```

```
DISPLAY A LINE ROUTINE
        MOVW    @A0,#9          *FUNCTION CODE
        MOVW    6@A0,#18        *LENGTH OF MESSAGE
        MOVW    8@A0,#H'0D      *TERMINATOR CHARACTER
        MOVL    10@A0,#MESSAGE1   *LOADS ADDRESS OF TEXT
        BRK     #0
JUMPDOS
        MOVW    @A0,#264        *FUNCTION CODE
        BRK     #0
MESSAGE1    TEXT    'INVENTORY PROGRAM'
TERM        DATAB   13              *CARRIAGE RETURN
SVC BLOCK
        RDATAB  32,0
        END     START
```

BRANCHING AND JUMPS WITH CALL AND RETURN

When writing programs in the BASIC language, the reserved word GOSUB branches execution to another section in the program where the desired instructions are carried out. When the command RETURN is encountered, execution jumps back to the next instruction following the GOSUB statement which sent it there.

Similarly, the instruction set of many microprocessors includes the instructions CALL and RET (for return), which perform the identical functions. The CALL statement causes the computer to jump to the location indicated by the label in the operand column. The editor/assembler manual that accompanies the Model 16 fails to mention that CALL also pushes the address of the next instruction onto the *stack*, an area in RAM used for temporary storage. The computer needs to save this location so that upon its return from executing the subroutine, it will know where it left off and can continue running.

The manual does correctly state that RETurn branches execution back to the instructions immediately following the CALL instruction. This is done by popping the stored address location off of the stack and placing it into the PC, or program counter register. This register always points to the memory address that contains the next instruction the computer is to carry out as soon as it completes the current line.

So, the CALL and RET instructions are in keeping with traditional microprocessor assembly-language programming methods.

The next sample program uses a CALL to branch to a routine to clear the screen and then jump back to the position cursor line. The program itself is somewhat similar to the previous one which cleared the screen, positioned the cursor, displayed a line of text, and returned to the DOS ready mode. In it, the section given the label CLEAR SCREEN is used as a subroutine. It is CALLed first to

get rid of all characters on the video display. The purpose here is to demonstrate how CALL and RET work in their simplest form. The reason for making the clear screen routine a subroutine is so that we can access it from various places within our program. By setting it up as a subroutine, we only need to include that routine once even though it may be used many times.

```
START        CALL           CLEAR SCREEN
POSITION CURSOR
             MOVW           @A0,#10           *FUNCTION CODE
             MOVW           6@A0,#12          *POSITION - ROW
             MOVW           8@A0,#40          *POSITION - COLUMN
             BRK            #0
* DISPLAY A LINE ROUTINE
             MOVW           @A0,#9            *FUNCTION CODE
             MOVW           6@A0,#16          *LENGTH
             MOVW           8@A0,#13          *ASCII TERMINATOR
             MOVL           10@A0,#MESSAGE1
             BRK            #0
JUMP BACK TO DOS
             MOVW           @A0,#264          *FUNCTION CODE
             BRK            #0
SVC BLOCK
             RDATAB         32,0
MESSAGE1     TEXT           'MIDDLE OF SCREEN'
             DATAB          13                *TERMINATOR
************************************************************
* CLEAR THE SCREEN SUBROUTINE
************************************************************
CLEAR SCREEN
             LDA            .A0,SVC BLOCK
             MOVW           @A0,#8            *FUNCTION CODE
             MOVW           6@A0,#27          *CODE TO CLS
             BRK            #0
             RET
             END            START
```

CALCULATING THE NUMBER OF CHARACTERS USING THE TEXTC OP CODE

If during an entire program only one or two lines of text need to be displayed on the screen, the routines we have discussed so far would be sufficient. However, it is more than likely that a program will require many messages and prompts to be shown. Since the display a line and the print a line (discussed in Chapter 7) routines need to know the number of characters in each message line, we must somehow devise a way of calculating the length of each line. One such method might be to create a table within the program. In it we could store all of the values necessary to indicate the length of each message. We could then take the number stored there and use it in each of the routines.

Perhaps an easier path to take is to use the pseudo-op code TEXTC. As you will recall, TEXT creates a block of memory and

stores a string of characters in it. The directive TEXTC does the same thing and more. It automatically calculates the number of characters in the line and stores this value as the first element in the string. The following program shows how this method can be used.

```
START
CLEAR SCREEN
          LDA       .A0,SVC BLOCK
          MOVW      @A0,#8  *FUNCTION CODE-OUTPUT CHAR TO VIDEO
          MOVW      6@A0,#H'1B   *ASCII CODE TO CLEAR SCREEN
          BRK       #0
POSITION CURSOR
          MOVW      @A0,#10    *FUNCTION CODE-POSITION CURSOR
          MOVW      6@A0,#1    *ROW POSITION (TOP TO BOTTOM)
          MOVW      8@A0,#38   *COLUMN POSITION (LEFT TO RIGHT)
          BRK       #0
DISPLAY A LINE  ROUTINE
          MOVW      @A0,#9    *FUNCTION CODE-DISPLAY A LINE
          MOVB      6@A0,#0   *INSURES A ZERO IS THERE
          MOVB      7@A0,/MENU  *LOAD # IN MENU INTO SVC BLOCK
          MOVW      8@A0,#H'0D  *LINE FEED-TERMINATOR CHARACTER
          MOVL      10@A0,#MENU+1  *LOAD ADDR. OF 1ST CHARACTER
          BRK       #0
JUMPDOS
          MOVW      @A0,#264  *FUNCTION CODE-RETURN TO DOS
          BRK       #0
SVC BLOCK
          RDATAB    32,0
MENU      TEXTC     'MENU'
          DATAB     13    *LINE FEED-TERMINATOR CHAR. (H'0D)
          END       START
```

There is some new food for thought in the above program. Since the label MENU now points to the first element, which is the number 4 because of the action of TEXTC, we can use this label to point to the length of the line of text. In its present form, the TEXTC command will store the value representing the length in one byte. Normally, byte-offset six and seven are used to store the length. The MC68000 requires values to be in the order of most significant byte, least significant byte. So, only byte-offset seven needs to have the value placed in it. Just to be sure that byte-offset six has no effect on the operation, we dumped a zero in it.

The terminator character is loaded into byte eight of the SVC block.

Next the address of the location storing the string of characters is needed. We cannot use MENU to point to the beginning of the text since the number 4 is now there, representing a "count" (thus the suffix "c" in TEXTC) of characters in the line. The actual location is the address of MENU plus one. The assembler is capable of doing this calculation right within the operand itself:

```
          MOVL     10@A0,#MENU+1
```

In the next chapter we will explore yet another way to fill the entire screen with text. Now we are beginning to combine many simple routines into more practical modules. Chapter 6 will show how text and keyboard input routines can work together.

Chapter 6
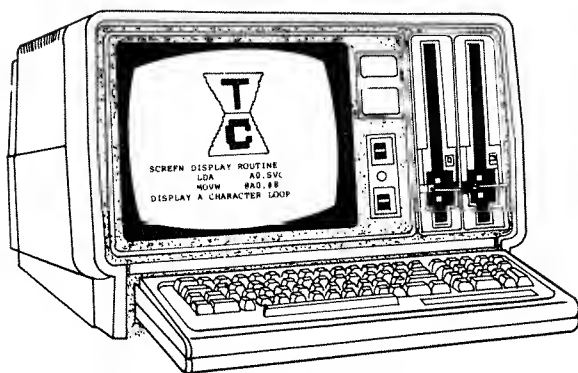
Keyboard Routines

It is extremely rare for a program of any real complexity not to need data from the operator. The Model 16s disk operating system contains two routines for inputting information from the keyboard. This information can then be employed within our program. You may develop a love/hate relationship with these two routines since they have both positive and negative characteristics.

A third routine is available to empty the buffer area in RAM which stores any keystrokes that the operator may have previously entered. Since the routine is usually executed prior to any keyboard input calls, we will cover it first.

CLEARING THE TYPE AHEAD BUFFER

A unique feature of the Model 16 computer is its ability to accept keyboard input from the user even though it is busy performing another duty. For example, when you first turn the machine on, you can type the date even before the screen prompts you with ENTER DATE (MM/DD/YYYY). The information you type is not immediately displayed on the screen, but it is stored in a buffer area. The computer will display and process that information when it catches up.

It doesn't take long before you begin to grow fond of the machine's ability to do this. You quickly learn that it is possible to enter quite a few commands well in advance. Of course, you must know what questions are coming up.

There may be times when you want to clear out this buffer and get rid of any extraneous keystrokes or input from the operator. A logical place for such a routine would be at the beginning of a program.

The disk operating system has a routine to perform this duty. The only value needed to be placed within the SVC block before calling it is the function code itself, which is one.

```
              LDA      .A0,SVC  BLOCK
              MOVW     @A0,#1
              BRK      #0
   SVC  BLOCK
              RDATAB   32,0
```

KEYBOARD CHARACTER ROUTINE

The Model 16 has the ability to accept one character or a string of characters from the keyboard. These can be either characters or control codes.

There are many instances when it is desirable to input one single character from the keyboard. For example, let us assume that you are writing a mailing list program and a menu is displayed on the screen. In all likelihood it would probably prompt the user to select a letter in order to branch to a section of the program where a specific task will be accomplished. The options might be something like this:

<A> ADD A NAME TO THE LIST
<D> DELETE A NAME FROM THE LIST
<P> PRINT MAILING LABELS
<S> SEARCH FOR A NAME ON THE LIST

SELECT A LETTER: A, D, P, OR S

In our sample menu above, it is preferable to simply have the user hit one letter and not have to hit the <ENTER> key. By typing only one keystroke, the operator can speed through a program more efficiently.

Two values must be placed within the SVC block of the routine that inputs a single character. The function code number is four. In byte-offset six and seven we must place a one or a zero to tell the machine to scan the keyboard once or to examine the keyboard and wait until a key is struck before returning to the assembly-language program. A value of one indicates a wait condition, and a zero instructs the machine not to wait.

For those readers who have programmed in BASIC on other models of Radio Shack computers, we make the following analogies.

When a zero is placed in the "wait/don't wait" section of the SVC block, the program does not stop as it would on a BASIC INPUT statement. If we consider the call as a subroutine, the BASIC equivalent might be represented by

```
100 A$=INKEY$
110 RETURN
```

A value of one placed in byte-offset six and seven prepares the routine to wait until a key is pressed. Again, this concept in BASIC could be written

```
100 A$=INKEY$
110 IF A$="" THEN GOTO 100 ELSE RETURN
```

Any keystroke will return it to the program.

This routine is easy to use. It does not print the letter chosen on the screen, just as it would not in our BASIC version above. In our fictitious mailing list program, it would not be necessary to display (referred to as "echo" back) the letter selected. The ASCII code of the character keyed in by the operator is stored in byte-offset eight and nine. We can then pull that value out of the supervisor block and use it within our program. The display a character routine listed in Chapter 5 could be used to print the character that has been entered on the screen. Thus, the routine to get a single keystroke from the keyboard would be set up like this:

```
          LDA      .A0,SVC BLOCK
          MOVW     @A0,#4            *FUNCTION CODE
          MOVW     6@A0,#1          *WAIT CONDITION
          BRK      #0
SVC BLOCK
          RDATAB   32,0
```

KEYBOARD LINE ROUTINE

Next we will explore the supervisor routine which allows the operator to enter multiple characters, words, and phrases for use within a program. The bare bones of the routine follow.

```
   LDA      .A0,SVC BLOCK
   MOVW     @A0,#5    *FUNCTION CODE
   MOVW     6@A0,#5   *MAX # OF INPUT CHARACT. ALLOWED
   MOVL     8@A0,#KEYBOARD STORAGE AREA
   BRK      #0
```

```
SVC BLOCK
        RDATAB    32,0
KEYBOARD STORAGE AREA
        RDATAB    5,0
```

Note that a long word is used to MOVe the address location defined by the label KEYBOARD STORAGE AREA. Four bytes are needed when dealing with all memory addresses in the Model 16, since the computer can handle up to 512K of RAM.

The keyboard line routine displays a row of periods on the screen while awaiting input from the operator. The number of periods is dependent upon the value placed in byte-offset five and six of the SVC block. This number specifies the maximum number of characters the computer is to accept from the keyboard for this particular input. This feature can be a plus, especially when the information entered by the user is to be placed in a disk file. Records in files must be set up in fields or sections, each having a predetermined length. It is a nice feature in that it shows the operator exactly how much space with which he has to work. Let us assume the user is asked to enter a part description on an inventory program. If the field in the record to be placed on the disk and the record has been set up to only allow room for a maximum of ten characters, the number ten can be placed in byte-offset five so that the user cannot attempt to enter more information than will be stored. In this way, he knows exactly what characters will be truncated when his response is stored on the disk. As the operator types in letters, the periods are replaced by the characters he enters.

While we feel that a keyboard input routine should limit itself to the job of gathering information from the operator to be used within the program, apparently the designers of the disk operating system did not. When an input is terminated, either by the operator hitting the <ENTER> key or by his typing in the maximum amount of characters, this routine will clear the rest of the screen following the last character entered. This clearing starts at the end of the text entered by the operator and proceeds to the end of the screen. The developers of the disk operating system made the decision for us. Personally, we do not like this mandatory "erase to the end of screen" feature, as the assembler manual refers to it. If the programmer has set up a nice-looking video display, perhaps with graphic designs around the borders or some other information being displayed on the screen below his keyboard input line, executing this supervisor call would be disastrous to his screen format. Such a feature can be either a blessing or a curse, depending on the

particular situation. For whatever our opinion is worth, we feel it is a drawback and creates more work for the programmer who must design his screen displays around it.

DISPLAYING A MESSAGE AND GETTING INPUT

The need often arises to have a message displayed on the screen and then wait for a response from the user. By examining the supervisor routines that we have covered so far, one might suggest first using the routine to display a line of text and then execute the keyboard line routine just demonstrated above. Well, we can applaud the disk operating system writers here for some foresight. They have given us a supervisor routine which calls both of these routines in a logical order.

```
          LDA      .A0,SVC BLOCK
          MOVW     @A0,#12   *FUNCTION CODE
          MOVW     6@A0,#26  *# OF CHAR. IN PROMPT MESSAGE
          MOVW     8@A0,#5   *MAX # OF INPUT CHAR. ALLOWED
          MOVL     10@A0,#KEYBOARD INPUT
          MOVL     14@A0,#DISPLAY PROMPT
          BRK      #0
SVC BLOCK
          RDATAB   32,0
KEYBOARD INPUT
          RDATAB   5,0
DISPLAY PROMPT
          TEXT     'ENTER THE AMOUNT OF LOAN $'
```

The function code is five and is loaded into byte-offset zero and one as usual. Offset six and seven hold the number of characters contained in the prompting message. The maximum number of characters that the operator is allowed to enter in is stored in eight and nine. The address where we wish to place the information entered by the user is loaded into byte-offset 10, 11, 12, and 13. As we have previously stated, it takes a long word to store an address for this computer. Similarly, the address of the text to be displayed is stored in 14, 15, 16, and 17.

TESTING FOR A MATCH USING THE CMP OP CODE

In the BASIC language, true/false testing is done by the IF-THEN statement. IF something is true THEN the machine is told to take a specific action.

Similarly, in assembly language we need a way to compare or check two values for equality. The op code CMP, or CoMPare, basically enables us to test various registers in the microprocessor

against a numerical value, another register, or the contents of an address in memory.

Compares are actually treated by the computer as a subtraction process. The "Z" or flag bit in the status register is set if the result of the subtraction is zero, that is, if it is a match. In Chapter 9 we will deal with the status register in more depth.

The next statement to follow this would logically be an instruction which tests the result of that comparison and acts accordingly depending upon the results. The BRanch command does just that.

THE BRANCH OP CODE

The status of the Z flag bit in the status register can be tested with the BRanch instruction. This is invaluable when you want to branch execution to another area of the program if certain conditions are met.

Next we will show a program which incorporates the display and keyboard routines we have discussed in this and the previous chapter. In it we make use of the CoMPare and BRanch instructions.

The program listing is given at the end of this chapter. We have included line numbers to make it easier to follow our explanations. First the screen is cleared. The cursor is then positioned to the screen location where we want the printing to begin.

The big job is displaying many lines of text on the screen. Here we put the display a character supervisor routine to work. Near the end of the program, the pseudo op TEXT establishes the many message lines we need to call up. DATA is used to define bytes of memory where we can place an ASCII code of 13. These are located between the lines of text where carriage returns and line feeds are needed.

Our technique here is to place the display a character routine inside a loop and print out all of the desired text. As previously shown in Chapter 5, this supervisor call requires that the ASCII code of the letter we wish to send to the display be placed in byte-offset six and seven. Since the character to display will be different with each pass of the loop, we need some way of constantly changing the character in offset six and seven.

Instead of placing the actual value there, we have chosen to use a label to point to the memory address where the ASCII code for the character we wish to display is located. The label STORE1 in our program is a four-byte area within the program that we use to store away the address of the next character to be sent to the screen.

All of the text to be printed is defined at the bottom of the program. The label MENU is used to indicate the address of the first character to be displayed.

Before entering the loop, we must place the memory address of MENU into the STORE1 area. This is done in a section labeled INITIAL SETUP. The address of the first character is loaded into a register; we selected A3.

```
LDA      .A3,MENU
```

Then the address stored in A3 is placed into our designated storage area.

```
STL      .A3,/STORE1
```

The slash (/) in front of STORE1 instructs the assembler that the address stored in register A3 is to be loaded into the memory location identified by the label STORE1.

Next the address of the SVC block is loaded into register A0 and the function code is placed in the appropriate position within the block.

The loop begins at line 18 and ends at line 28. Once inside the loop, the ASCII code located at the memory address which has been saved in STORE1 is loaded into register A3. This takes place in line 19. The character in A3 is moved into byte-offset seven of the SVC block. Since we are only dealing with one byte, we chose to place the character directly into the SVC block. Since the MC68000 stores values in the form of most significant byte, least significant byte, the ASCII value (being between 0 and 255) is placed in offset seven and not six. To be sure that there will not be any chance of error, we opted to load a zero into offset six.

In order to prepare for the next character to be displayed, the memory address stored in A3 is incremented by one.

```
ADDW     .A3,#1
```

This new address is stored away in our storage area, ready to be called up the next time a character is to be sent to the screen.

```
STL      .A3,/STORE1
```

Now the supervisor call is made in line 24.

We must test this byte to see if we have reached the end of our text. At the end of the defined text we placed a terminator code. There are several ASCII codes which the Model 16 owner's manual

indicates have no purpose and are unused by the machine. We chose a value of five which is one of these unused numbers. It is a good idea to avoid using a character since there may be an occasion when it would be necessary to display that particular character. A test can now be made on each character to see if it is the terminator character.

We have not yet used any of the data registers in our program. It is easy to compare the value in a data register with a numerical value using the CoMPare instruction. So we load the value of the character (which has been placed in byte-offset seven of the SVC block) into a data register, such as D0. Prior to that we place all zeros in D0 to insure there are no other values present that might creep into our program and give us trouble.

```
LDL        .D0,#0
LDB        .D0,#5
```

The   CMPB   .D0,#5   instruction compares a byte, that is, it subtracts the number in D0 from five. If the two numbers are not equal, the computer is directed to branch execution back to the beginning of the loop and get another character to display. BNE represents "branch if not equal" to the location indicated in the operand.

```
            CMPB       .D0,#5
BNE         DISPLAY A CHARACTER LOOP
```

If a match is found, execution falls through to the next instruction, which is on line 29. This is the keyboard input routine described earlier in this chapter. The computer waits for the operator to type one keystroke. The operator's answer is stored within the program in the space identified by KEYBOARD STORE1. We used RDATAB to define KEYBOARD STORE1 and STORE1 storage areas, but DATAB would also have done the same thing.

Again by using the CMP op code, the character entered by the operator can be tested for an "A" or "S" response. BE is a "branch if equal" instruction which sends program execution to the proper section. If the user selects a letter other than "A" or "S" he is returned to the disk operating system ready mode.

We have not included "add" or "search" sections to this particular program, as the intention here is to present the concepts of displaying text and comparing values as clearly as possible. Chapter 10 develops a complete program showing how these sections might be written. Here is a listing of the full keyboard routine using the subroutines we've discussed in this chapter.

```
 1 *THIS PROGRAM DEMONSTRATES CREATING A FULL SCREEN MENU
 2 START
 3 CLEAR SCREEN
 4         LDA     .A0,SVC BLOCK
 5         MOVW    @A0,#8  *FUNCTION CODE-OUTPUT CHAR.
 6         MOVW    6@A0,#H'1B  *ASCII CODE TO CLEAR SCREEN
 7         BRK     #0
 8 POSITION CURSOR
 9         MOVW    @A0,#10  *POSITION CURSOR FUNCTION CODE
10         MOVW    6@A0,#1  *ROW POSITION (TOP TO BOTTOM)
11         MOVW    8@A0,#38  *COLUMN POSITION (LEFT-RIGHT)
12         BRK     #0
13 INITIAL SETUP
14         LDA     .A3,MENU
15         STL     .A3,/STORE1  *STORES AWAY BEG.MENU ADDR.
16         LDA     .A0,SVC BLOCK
17         MOVW    @A0,#8  *FUNCTION CODE—DISPLAY A CHAR.
18 DISPLAY A CHARACTER LOOP
19         LDL     .A3,/STORE1
20         MOVB    7@A0,@A3
21         MOVB    6@A0,#0  *INSURES A ZERO IS THERE
22         ADDW    .A3,#1  *INCREMENT-POINT ADDR.NEXT CHAR
23         STL     .A3,/STORE1  *STORE AWAY NEXT CHAR. ADDR
24         BRK     #0
25         LDL     .D0,#0
26         LDB     .D0,7@A0
27         CMPB    .D0,#5
28         BNE     DISPLAY A CHARACTER LOOP
29 KEYBOARD INPUT ROUTINE
30         MOVW    @A0,#5  *FUNCTION CODE
31         MOVW    6@A0,#1  *# OF CHARA. TO BE ENTERED
32         MOVL    8@A0,#KEYBOARD STORE1
33         BRK     #0
34 *TEST FOR AN "A" OR "S"
35         LDB     .D0,/KEYBOARD STORE1
36         CMPB    .D0,#65  *TEST FOR AN "A"
37         BE      ADD ROUTINE
38         CMPB    .D0,#83  *TEST FOR AN "S"
39         BE      SEARCH ROUTINE
40 JUMPDOS
41         MOVW    @A0,#264  *FUNCTION CODE
42         BRK     #0
43 ADD ROUTINE
   -- program continues here --
44 SEARCH ROUTINE
   -- program continues here --
45 SVC BLOCK
46         RDATAB  32,0
47 STORE1
48         RDATAB  1,0
49 KEYBOARD STORE1
50         RDATAB  1,0  *STORAGE AREA FOR USER INPUT
51 MENU    TEXT    'MENU'
52         DATAB   13,13
53         TEXT    '========================================='
54         TEXT    '========================================='
55         DATAB   13,13,13,13
56         TEXT    '<A>   ADD A NAME TO THE LIST '
57         DATAB   13,13
58         TEXT    '<S>   SEARCH FOR A NAME'
59         DATAB   13,13,13,13
60         TEXT    'SELECT THE APPROPRIATE LETTER  > '
61         DATAB   5    *TERMINATOR CHARACTER
62         END     START
```

Chapter 7
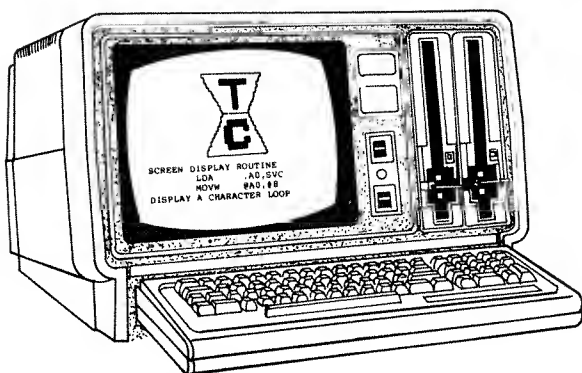
Directing Output to a Printer

Similar to the routines that direct output to the video display, the disk operating system contains routines for printing information on the printer. Two routines are given for sending characters to the printer while two additional calls are used to control the printing format of each page. The latter routines test or set such printer parameters as the number of lines on a physical page, the logical length of the page, and the number of characters printed on one horizontal line. These two routines are fairly self-explanatory and are well covered in the Assembler-16 manual, so we will not be dealing with them here.

PRINT A CHARACTER

Chapter 5 discussed two video routines, one designed to output a single letter or code to the screen and the other to display a string of characters. Chapter 6 similarly demonstrated a routine to get one character from the keyboard and a routine that gets an entire line. As you might have guessed, TRSDOS provides us with two similar routines for directing output to a printer.

The print a character call is straightforward and does not require any special treatment. We should mention that since most printers do not actually print until their buffer is filled or until a line feed code is sent to them, the routine below may not give immediate action on your printer. Fig. 7-1 shows the printer for the Model 16. In our example, the letter "A" is sent to the printer.
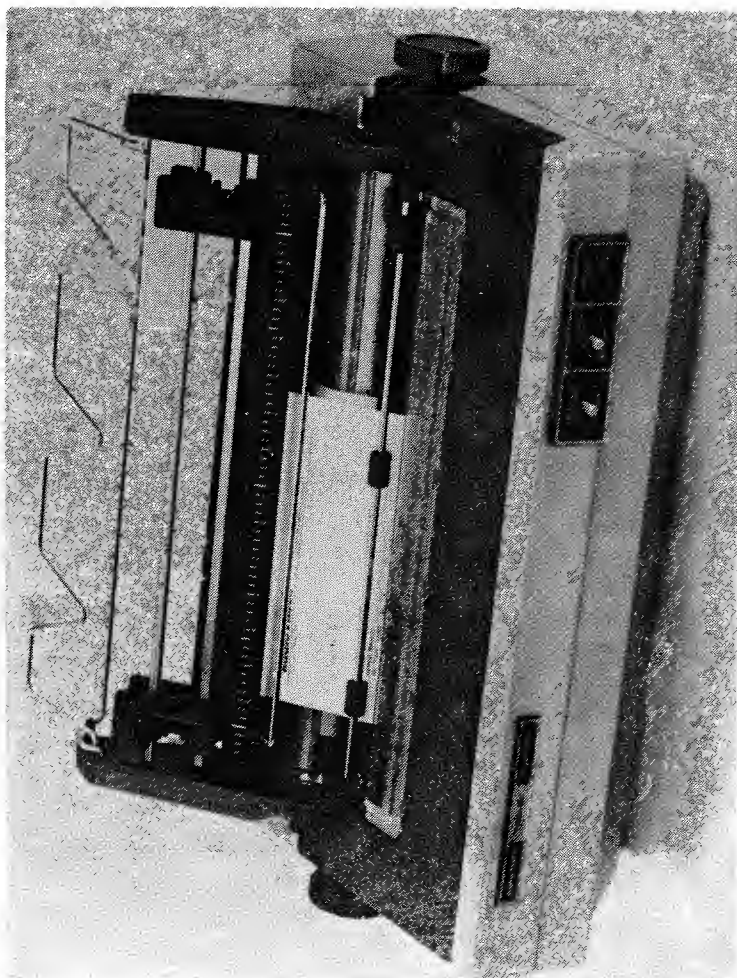
Fig. 7-1. Radio Shack's newest daisy wheel printer, one of many printers offered by Radio Shack for use with the Model 16 and other microcomputers in their line. (Courtesy of Radio Shack)

The function code for this call is 18.

```
        LDA     .A0,SVC BLOCK
        MOVW    @A0,#18   *FUNCTION CODE
        MOVW    6@A0,#65  *ASCII CODE-LETTER "A"
        BRK     #0
SVC BLOCK
        RDATAB  32,0
```

In order to see if the routine is working, the simplest thing to do is to execute the routine twice, first by sending a character, then sending an ASCII code 13 (line feed/carriage return) which will cause the printer to empty its buffer.

```
        LDA     .A0,SVC BLOCK
        MOVW    @A0,#18    *FUNCTION CODE
        MOVW    6@A0,#65   *ASCII CODE FOR "A"
        BRK     #0
        MOVW    6@A0,#13   *LINE FEED/CARRIAGE RETURN
        BRK     #0
            SVC BLOCK
                RDATAB  32,0
```

PRINT A LINE

In order to print a line, the length of the line must be placed in byte-offset six and seven. The ASCII code of the character to be sent to the printer after the text is printed is placed in offset eight and nine. Similar to the display a line routine covered in Chapter 5, the owner's manual refers to this code in offset eight as a "terminator" character, although it does not terminate anything. It's not really the best phrase to describe the function. The actual termination of the printed message is determined by the number placed in byte-offset six which limits the number of characters to be sent to the printer.

The address of the text is placed in byte-offset 10, 11, 12, and 13 of the SVC block. Thus we must use the suffix, "L," on the MOVe command.

```
        LDA     .A0,SVC BLOCK
        MOVW    @A0,#19    *FUNCTION CODE
        MOVW    6@A0,#14   *LENGTH OF LINE TO BE SENT
        MOVW    8@A0,#13   *"TERMINATOR" CHARACTER
        MOVL    10@A0,#PRINTER MESSAGE
        BRK     #0
PRINTER MESSAGE
        TEXT    'INVENTORY LIST'
SVC BLOCK
        RDATAB  32,0
```

Chapter 8
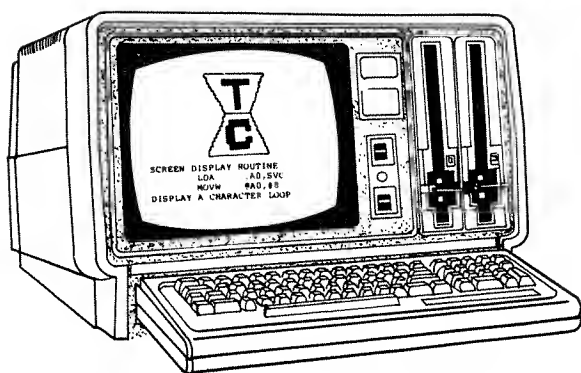
Disk Input and Output Routines

When we write computer programs, the instructions which make up the program reside in the computer's RAM. The microprocessor fetches one instruction at a time from RAM and carries out the orders. Programs are placed in memory by loading them from some type of storage medium such as a disk, or by typing them in via the keyboard.

RAM can also be used as a storage area for various items. However, RAM loses all information stored there when the power is turned off; it can only be used to temporarily store data.

To permanently store information, we must place the data on disk. Basically there are two kinds of disk storage devices, the floppy disk and the hard disk. A *floppy disk* (also sometimes called a diskette) cannot store as much information as a hard disk can, but it has the advantage of being easily swapped between other computers of the same model. Also floppy disks can be economically carried or sent through the mail.

On either type of disk, we can create files. Disk *files* are just like the files you would find in any place of business; in either case the basic concept is the same. By comparing a disk file to a typical index card file, we can more easily explain some terms.

A file is one entity, made up of many records. In an index file, each card is considered to be one record. Each record can be broken down into smaller entries called *fields*. Every record contains the same fields. Suppose we have an index card file that gives us a list of

names and addresses of all the charge customers for a particular business. Each record would contain the same fields, that is, each would have a field for the name, one for street address, town, state, and zip code. Every record is made up of the same fields, but naturally each record will contain different names and addresses.

In this chapter we will discuss the fundamental routines needed to create and maintain a disk file. By "maintain" we refer to the adding of new records to an existing file or the modifying of existing records.

THE FLOPPY DISK

The Model 16 computer is capable of using either single-or double-sided disks. Single-sided disks record information only on one side; double-sided disks store data on both sides. The Model II computer, however, can only access one side of a disk. When a Model II is upgraded to become a Model II/16, that is, a Model II with the MC68000 circuitry added to it, the disk drives are not touched. Therefore the Model II/16 cannot utilize two sides of a disk either. (Chapter 11 discusses the Model II/16 in more depth.)

The Model 16 has two "photo eyes," which are used to detect whether a disk is single or double sided. Double-sided disks simply have both sides rather than just one side coated with a ferromagnetic material.

The placement of the indexing sector hole, which is present on every disk, is used by the machine to detect whether the current disk in the drive is single or double sided. The indexing hole is placed in a slightly different location on the two types of disks. The photo eyes, also known as photo cells, are electronic devices that detect light. Inside the disk drive mechanism light bulbs are placed on one side of the disk. Two photo cells are located at the position where they will line up with the proper indexing hole. Should one photo cell see light, then that might indicate a single-sided disk. If the other photo cell sees the light, then it would mean the disk could be accessed on both sides.

Figure 8-1 shows both a single-sided and double-sided disk. Note the few degrees difference between the indexing sector holes.

The indexing hole aligns with the photo cells once every revolution. The computer also uses this hole when formatting a disk (with the FORMAT command). Track and sector information is set up on the disk when it is formatted and the alignment of the indexing hole gives the machine a reference point on the disk.

Tracks are laid out in concentric circles. Each track, except for

Fig. 8-1. Single- and double-sided disks. Note the difference in placement of the indexing sector holes. (Photograph by George M. Keen)

track zero on side one, contains the same number of bytes. As the tracks approach the center of the disk, the data is jammed closer together. Mechanical and electrical precision are needed to insure the proper reading and writing of information in these higher numbered tracks. Track zero is located on the outside rim of the disk and track 76 is closest to the center, making a total of 77 tracks.

The disk operating system of the Model 16 will support either single- or double-sided disks. The disk itself determines whether or not both sides will be accessed. Double-sided recording is transparent to the operator. The machine keeps track of all of the information stored on the disk. We cannot specify on which side of the disk we wish to store information. A single-sided disk is formatted to have 77 tracks. As far as the user is concerned, a double-sided disk appears to him as one disk with 154 tracks (77×2).

Each track consists of 32 sectors. A sector is made up of 256 bytes. A Model 16 disk contains a total of about 625,290 bytes per side. If a double-sided disk is used, the total number of bytes is 1,256,704. The reason two-sided disks do not have exactly twice as many bytes as one-sided disks is because track zero on side one is formatted differently than the rest. It is used exclusively by the disk operating system. While the other tracks are formatted in the "double-density" mode, this "boot" track is single density. The machine apparently is in the single-density mode when it is turned on and the information on this track kicks it into the double-density

Hardware	Sides	Sectors per disk	Tracks per disk	Total bytes
Model I	SINGLE	350	35	89,600
Model III	SINGLE	720	40	184,320
Model II	SINGLE	1,875	77	509,184
Model II/16	SINGLE	1,875	77	509,184
Model 16	DOUBLE	4,928	154	1,256,704
5M HARD DISK	-	19,584	612	5,013,504
8M HARD DISK	-	34,816	1,024	8,912,869
12M HARD DISK			1,836	11,898,400

Fig. 8-2. Amounts of disk capacity on the various Radio Shack microcomputers.

mode. *Single* and *double density* are terms which describe the technique used to store information on a disk. Double density packs more data onto the same size disk than single density, thus giving it increased storage capacity.

The Model II/16 formats disks a little differently and thus has a total storage capacity or 509,184 bytes per disk.

Not all of these bytes are available to the user. The disk operating system requires some of the tracks to store its routines. Figure 8-2 lists the various amounts of disk capacity on the most popular Radio Shack model microcomputers, the 5 megabyte hard disk drive (shown in Fig. 8-3) and the 8 megabyte hard disk storage devices. The chart shows the impressive storage capability of the Model 16. Radio Shack also has a new 12M hard disk.

THE HARD DISK

The Radio Shack 8 Megabyte Hard Disk storage device is designed to be connected to the Model II, 16, and II/16. Unlike floppy disks, the disks used here are invisible to the user. They cannot be taken out of the unit. Inside the square-shaped case are two platters or "disks." Each disk is double sided making a total of four recording surfaces. While the disks are sectioned off into tracks and sectors similar to floppy disks, an additional factor called a "cylinder" comes into play.

On each of the four surfaces, the same number of tracks is layed out and numbered. The track numbers are located at the identical positions on the disk. Figure 8-4 shows a three-dimensional view of the platters. Electronic switching selects which head is to be accessed. A *head* is a small device made up of a coil of wire which can transfer electrical signals to and from a magnetic storage

Fig. 8-3. Pictured is Radio Shack's 5 megabyte hard disk drive. (Courtesy of Radio Shack)

47
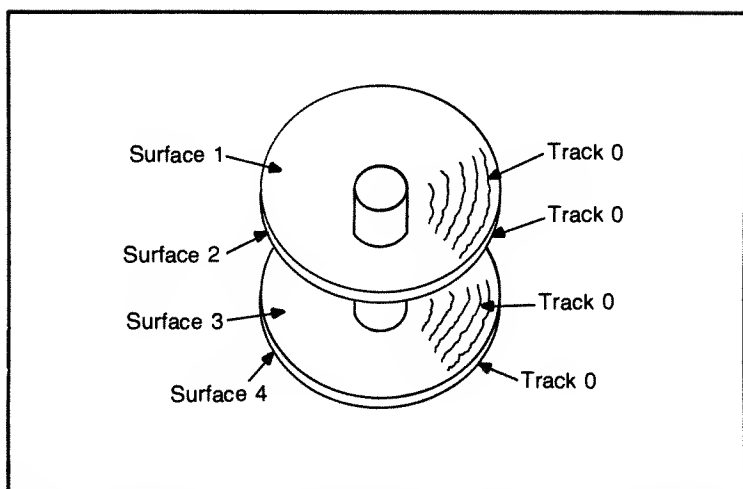
Fig. 8-4. A three-dimensional view of the two disk platters in a hard disk drive. The relationship of tracks on each surface can be seen. Track zero on the top of the first disk, for example, is located at the same distance from the center of the disk as track zero on the underside surface as well as track zero on the top and bottom of the second platter. (Graphic art by Betty Tocci)

medium. The concept is similar to the record and playback heads in tape recorders.

Each head can be used to either read information from the disk or write data to it. The read or write function is selected electronically by the machine.

The four read/write heads are all positioned over the same track number on all four surfaces. The ability to switch electronically between all four heads and tracks without physically moving any parts helps account for the great speed at which hard disks can store and retrieve information.

For example, track ten on the top disk has the same radius as track ten on the bottom disk. The track located on the outer rim is track zero. Similarly, track zero on the bottom of the same platter is located directly underneath track zero on the top. Track zero on the top and bottom of the second disk is also in the same physical position.

These four tracks over which the heads are positioned at any given time comprise one unit called a cylinder. There are 34 sectors per track, so four tracks contain a total of 136 sectors. This makes up one cylinder. Figure 8-4 shows a three-dimensional view of the two platters and the relationship of the same track number on each platter.

OPENING A FILE

The concepts of opening a file, getting or putting a record onto the disk, and using a buffer as a temporary storage area where information passes on its way to and from the disk, are the same as in the BASIC language.

BASIC does a lot of things for us. In assembly programming we are required to keep track of some of the parameters which BASIC normally handles for us.

The first thing that must be done to create a file where we can store information is to open a file on the disk. This is accomplished with the OPEN statement. In an office situation, it is necessary for you to first "open" the drawer of the file cabinet before you can begin to place any records in the file.

TRSDOS allows us to specify several ways in which we may choose to open a file. We may wish to create a file that has never before been in existence. The first time we open the file to place data into it is the time to use this creation option.

We may want to open a new file but not if it already exists, since valuable information currently stored there may be destroyed if we overwrite old records with new ones.

Another type of open condition would be to open a file only if it has previously been created.

At the time we open a file, these and other parameters must be set up to instruct the computer how to go about opening the file we have chosen.

The open supervisor call has an identifying number of 40. A name must be given to the file to identify it from other programs and files on the disk. TRSDOS file names can consist of up to eight characters. The first character must, however, be a letter of the alphabet. The remaining seven can be any alphanumeric character, that is, letters or numbers. Generally you should avoid using symbols and punctuation when naming a file. Frequently programmers will use an extension of three letters which the disk operating system allows. If the eight characters are followed by a slash (/), then an extension of three more letters is permitted. Typically a /CMD is placed at the end of a file name to signify a command program, or a machine-language program. To represent a data file such as a mailing list program, the extensions /DAT are commonly used. File extensions are optional. An important note: please be advised that we ran into trouble when using the extension /FIL so we advise avoiding it to represent "file."

The list of names which our mailing list program will create

could be stored in a file called something like LIST/DAT to show that the LIST file name is not a program but rather a data file.

Entry conditions before calling the open supervisor routine require the usual function code to be placed in byte-offset zero of the SVC block.

The address where we specify the name of the file we wish to open is placed in byte-offset six through nine. Since this is an address, it requires four bytes. An ASCII code of 13 (carriage return) is needed at the end of the location where the file name is defined. The TEXT op code can be used to establish the file name and the DATAB instruction will reserve a byte of space within the program for the carriage return code.

Finally, offsets 10 through 13 store the address where we have established a five-byte block of memory that has been filled with instructions concerning the opening of the file. The Model 16 owner's manual refers to this block as a parameter list. Similar to the SVC block, five consecutive bytes in memory are used to establish certain criteria for the file.

In the first position, which is an offset of zero, we place a code representing how the file is to be accessed. Three combinations are possible, "R," "W," or "P." The access code "R" can be tagged on if we want to limit the user's access to the file. By being able to only "read" the information in a file, the user is kept from recording any data into the file. The "R" code, for "Read" access only, can be set up in the designated memory block by either using the EQUate op code or by directly placing the value into the desired location with a MOVe command. In a moment we will show how to perform that operation both ways.

Regardless of the method you use, probably the best procedure is to use a register as the temporary holding spot for the parameter list. We have been doing the same type of thing with register A0 and the SVC block throughout this book. But this will be slightly different. First, a register can be filled with the necessary code. Then the contents of that register can be placed into the memory area where we are storing these items prior to calling the open supervisor routine. In keeping with the owner's manual, we shall set up our open routines using basically the same program skeleton.

Since register A0 is tied up keeping track of the SVC block location, we can choose any one of seven other address registers. Decisions, decisions! How about if we use A1, but you can just as easily use any address register your little heart desires. Now we can look at the two ways of getting these codes into register A1.

First we will show how EQUate can be used to make life easy for us programmers. All we have to do is EQUate a byte of text. By putting quotes or apostrophes around the alphabetical character, the assembler will automatically convert it to the ASCII equivalent for us and plug in the correct value in register A1.

```
        MOVB      @A1,#READ/WRITE ACCESS CODE
          .
          .
```

(program continues)

```
          .
          .
READ/WRITE ACCESS CODE
        EQUB      'R'
```

It is much simpler to directly place the ASCII number for the desired character into the register itself as in

```
        MOVB      @A1,#82
```

In the following instruction, the programmer had to look up the ASCII code for the letter "R" which is 82 decimal, or a 52 hexadecimal.

```
        MOVB      @A1,#H'52
```

Personally, we feel that the latter two examples are preferable to the EQU method since it tends to keep the program shorter and cleaner. A comment or remark can always be added to the line to make it more self-documenting.

However, as with any type of programming, there isn't necessarily a right or wrong way to go about setting this thing up. We don't know who originally said it, but an analogy has been made between artists painting a picture and programmers programming. If you take ten artists and let them paint a picture of a sunset, you will get ten different pictures. Each has its own merits, but none is really right or wrong. The same holds true for programming. Under certain circumstances, one technique may be better than another, but as long as both perform equally well in the final mix, it doesn't really matter.

Getting back to the other two access codes, "W" and "P," we have "W" to allow both reading and writing of data files. "P" is used to permit the reading and writing of Z80 program files.

The next byte in the parameter list, byte-offset one, stores the length of the record. Record length can be up to 256 bytes.

Two types of files can be handled by TRSDOS; variable length files and fixed length files. Variable length is not as common as fixed files. It allows each record within a file to have a different length. In this book, we will be dealing with files which have predetermined record lengths that do not change from one record to the next. Just as in the case of the access code, letters are used to control the desired function, "V" for variable length and "F" for fixed length. This value is positioned in byte-offset two.

In byte-offset four the number 0, 1, 2, or 3 is placed to specify the conditions which we want to open the file. Figure 8-5 summarizes what each code represents.

The last item in the parameter list is the attribute code. The disk operating system supposedly pays no attention to this number. It is strictly for our own use. With it we can assign our own identification code to the file. Valid numbers are 32 through 255, but not zero as the owner's manual states. The number is placed in offset four of the list.

Once we have digested all of that material, we are ready to open a file.

In the Model 16 owner's manual is a partial listing found under the directive OPEN in the chapter on supervisor calls. Like OPEN, most of the routines are not intended to be complete programs, but rather short examples. Therefore they lack a BEGIN or similar label at the top of the program and an END instruction at the bottom, both of which are necessary in order to get the program to run on its own. Be advised that we were not able to get the routine to work exactly as it is shown, even with the addition of the END instruction. The five instructions that load five separate bytes of information into a block of memory set aside as the open routine's parame-

Code 0	Open the file. If the file has never previously been opened, do not open it now.
Code 1	Create the file. If the file already exists on the disk, do not open it. Return an error if the file has been created prior to now.
Code 2	Create the file. If the file already exists on the disk, destroy it.
Code 3	Open the file to have shared access. If the file has never previously been opened, do not open it now.

Fig. 8-5. Creation condition codes for files.

ter list, cause the machine to give an odd address trap error message when the program is run. Here is the listing from the manual:

```
MOVW        @A1,#WRITE ACCESS
MOVW        1@A1,#RECORD LENGTH
MOVW        2@A1,#FIXED FILE
MOVW        3@A1,#OPEN ONLY IF EXISTS
MOVW        4@A1,#USER ATTRIB
```

When the routine failed to work, we changed the OPEN ONLY IF EXISTS code (currently holding a value of 0) to a one, which should create a new file, according to the previous chart. Therefore, if the routine worked at all it would be easy for us to see since we could examine the directory from TRSDOS with the DIR command and see if file name had indeed been created on the directory.

After making that change, the program still did not work and continued to give an odd address trap error message (see Chapter 9 for an explanation of this error message). We reasoned that perhaps only a single byte should be indicated by the MOV instruction instead of a word, that is, MOVB rather than MOVW. It seemed logical since one byte is all that is being MOVed, anyway. After changing all of the MOVW instructions in the parameter list to MOVB, the routine worked fine.

As is our usual practice, we prefer to place numbers directly into registers and addresses whenever possible instead of using EQUate to define their values elsewhere in the program. This helps to keep the program shorter, clearer, and with less chance of reusing variable names that may be storing important information. Since comment lines take up no room in the final listing, documentation of the program to keep everything easily understandable can be used as much as necessary.

So our complete program to open a file on the disk such that you can see the file name on the directory looks like this:

```
OPEN    LDA     .A0,SVC BLOCK
        MOVW    @A0,#40             *INDENTIFYING FUNCTION CODE
        LDA     .A1,FILENAME
        STL     .A1,6@A0
        LDA     .A1,PARAMETER LIST
        MOVB    @A1,#80             *READ/WRITE ACCESS
        MOVB    1@A1,#100           *LENGTH OF RECORD
        MOVB    2@A1,#H'46          *FIXED/VARIABLE FILE
        MOVB    3@A1,#1             *CREATE A NEW FILE
        MOVB    4@A1,#32            *ATTRIBUTE CODE
        STL     .A1,10@A0
        BRK     #0
JUMPDOS
        MOVW    @A0,#264
        BRK     #0
```

```
SVC BLOCK
        RDATAB   32,0
FILENAME
        TEXT     'LIST/DAT'
        DATAB    H'0D              *CARRIAGE RETURN TERMINATOR
        DATAB    0                 *RESERVE 1 BYTE TO KEEP
                                   *INSTRUCTIONS ON EVEN
                                   *ADDRESS LOCATIONS
PARAMETER LIST
        RDATAB   6                 *ONLY 5 BYTES USED BUT 6
                                   *KEEPS INSTRUCTIONS ON EVEN
                                   *ADDRESS LOCATIONS
        END      OPEN
```

Let's carefully go through this routine. The first thing we do is to load the address of the SVC block into register A0 and move the identifying supervisor code into byte-offset zero and one.

Next we need to set up a block of memory and by using the TEXT directive establish the file name that we want on the disk's directory. This can be done in two moves. The location of the file name can appropriately be labeled FILE NAME. This address is loaded into register A1 and from A1 it can be moved into byte-offsets 6, 7, 8, and 9 of the SVC block where it must reside prior to calling the open supervisor routine. The two instructions are:

```
        LDA      .A1,FILE NAME
        STL      .A1,6@A0
```

In keeping with the Assember-16 manual, we will use the same idea of setting up the parameter list in one area of memory whose address location is pointed to by register A1. Just as we use A0 to point to a spot in memory where we can place the various values into it by using the byte-offset feature, we can address A1 indirectly with byte offsets, also. Once we have loaded the address of the parameter list into A1, each item in the list is placed into the proper position in the block one byte at a time. Offset zero stores the code telling the machine how we want to access the file. There are two possibilities—read only or both read and write. "R" is for read only access, "W" for read or write, and "P" is for reading and writing of Z80 program files. The decimal ASCII codes are 82 for "R," 87 for "W," and 80 for "P." These values can be placed directly into the desired offset. Here is the instruction to both read and write:

```
        MOVB     @A1,#80
```

Byte-offset one of the parameter list block indicates the length of the record in bytes. In this example we chose 100:

```
        MOVB     1@A1,#100
```

The type of file, fixed or variable, which is placed in offset position two, can have the ASCII value placed directly into it just as we did with the access code parameter. Here an "F" or a "V" is needed. The ASCII code for "F" is 70 decimal or 46 hexadecimal. In our sample listing we use the hexadecimal value simply to show how it is possible to use either decimal or hexadecimal values interchangeably. The ASCII for "V" is 86 decimal. The instruction for "F" or fixed file is

```
MOVB     2@A1,#H'46
```

The creation code, either the number 0, 1, 2, or 3 as shown previously on the creation code chart, establishes the conditions for opening the file. This value is placed in byte-offset 3 of the parameter list. We have used the code number one which creates the file:

```
MOVB     3@A1,#1
```

The final byte in the parameter list contains the user's attribute code. According to the manual, this number is for our own use and the computer does not look at it. However, it does affect the computer somehow since placing a zero there caused problems.

```
MOVB     4@A1,#32
```

Now that the parameter list has been set up, its address location is loaded into the proper positions in the SVC block, namely offsets 10, 11, 12, and 13.

With these entry conditions ready to go, the supervisor routine can be called.

At the bottom part of the program we have defined memory spaces for the SVC block, the file name, and the parameter list. The file name is fixed by using the TEXT pseudo op to define the alphanumeric characters and by terminating the phrase with an ASCII code of 13 for a carriage return. It is important that this 13 be put in place. We feel it is good practice to keep the length of instructions and block definitions to even numbers. This helps prevent possible odd address trap errors when the program is run. You may have noted that we defined an extra byte of memory after the carriage return. By the same token, we set aside six bytes for the parameter list, even though only five are needed.

```
FILENAME
         TEXT      'LIST/DAT'
         DATAB     H'0D
         DATAB     0
PARAMETER LIST
         RDATAB    6,0
```

Upon exiting from the open supervisor routine, any error codes are placed in byte-offset two and three as usual.

When the file is opened, TRSDOS assigns a number to the file to identify it. This number is returned in offsets 14 and 15. We need not be concerned about the value of this number. However, we do have to use this number whenever we want to perform any operations on this file such as reading or writing a record and closing the file. We will show how to handle this number next as we discuss writing a record to the file.

WRITING TO THE FILE

In order to write a record into the file we just opened, we use the supervisor routine number 44 entitled DIRWR in the Model 16 owner's manual.

As we have previously mentioned, the disk operating system assigns a number by which it can identify that particular file. This is returned as a word length in byte-offsets 14 and 15 in the SVC block. Whatever that value may be, it must be taken from byte-offsets 14 and 15 and placed into offsets 6 and 7 to prepare the block for entry into the write routine. This is accomplished by the following lines:

```
LDW     .A1,14@A0    *MOVE FILE ID # INTO A1
STW     .A1,6@A0     *MOVE FILE ID # INTO SVC BLOCK
```

Besides storing the function code of the write routine as well as the file identification number into the SVC block, two other parameters need to be set. The address which stores the data or text to be written to the disk is positioned in offsets 8 to 11. We have given this storage area or "disk buffer" a length of 256 bytes. That lets us use that area for other files of different lengths. The record length will be controlled by the open routine, anyway. Only as many bytes as were indicated at the time the file was opened will be written to the disk.

The last step is to place the record number into offsets 12 through 15. Unlike the BASIC on other Radio Shack microcomputers, the first record in the file has a number of zero. Zero is not a valid record number in BASIC. Records are numbered in consecutive ascending order, 0, 1, 2, 3, and so forth with the maximum number being 4,294,967,295 decimal or FFFFFFFF hexadecimal. That is a big file!

In a moment we will create a short program that will demonstrate the write routine.

CLOSING THE FILE

When we are done with a file, it must be closed. In an office, when a secretary is finished entering or examining records in a file cabinet, she closes the drawer.

The identifying function code for closing a file is 42. Only one other parameter need be set up and that is the file identification number which the machine created at the time the file was opened, as mentioned previously. That value is placed in byte-offsets 6 and 7. In this way the computer knows the specific file you want to close. It is possible to have more than one file open at any given time and you may want to close one but not another.

So with all of this knowledge we can get our feet wet with disk file concepts by writing a small program that opens a file, writes a string of characters (a bunch of "As" into the first record), closes the file, and returns to the TRSDOS ready mode.

To see if this program does create a file and fill it with "As" you can use the LIST command. After running the program from the TRSDOS ready mode, type LIST LIST/DAT. This will show you the file and the contents of its one record.

Program that opens a file, writes a character string, and closes file

```
START
OPEN        LDA       .A0,SVC BLOCK
            MOVW      @A0,#40
            LDA       .A1,FILENAME
            STL       .A1,6@A0
            LDA       .A1,PARAMETER LIST
            MOVB      @A1,#80            * ASCII FOR "P"
            MOVB      1@A1,#80           * RECORD LENGTH
            MOVB      2@A1,#H'46         * ASCII FOR "F"
            MOVB      3@A1,#1            * CREATE A FILE
            MOVB      4@A1,#32           * ATTRIBUTE CODE
            STL       .A1,10@A0
            BRK       #0
WRITE       LDA       .A0,SVC BLOCK
            MOVW      @A0,#44
            LDW       .A1,14@A0
            STL       .A1,6@A0
            LDA       .A2,RECORD BUFFER
            STL       .A2,8@A0
            LDL       .A3,#0             * 1ST RECORD, # 0
            STL       .A3,12@A0
            BRK       #0
CLOSE       LDA       .A0,SVC BLOCK
            MOVW      @A0,#42
            STW       .A1,6@A0
            BRK       #0
JUMPDOS
            MOVW      @A0,#264
            BRK       #0
SVC BLOCK
            RDATAB    32,0
```

```
FILE NAME
        TEXT      'LIST/DAT'
        DATAB     13
        DATAB     0
PARAMETER LIST
        RDATAB    6,0
RECORD BUFFER
RDATAB  256,65
END     STARTS
```

READING A RECORD FROM A FILE

The DIRRD routine, as it is referred to in the Model 16 owner's manual, is very similar to the write routine, only now we need to pull a record off the disk and place it into a buffer. Its function code is 35. This buffer address is placed into offsets 8 to 11. The file identification number goes into 6 and 7. The number of the record we wish to read is placed in 12 through 13.

Byte-offset 16 allows us to limit the users access to the record by locking it. A zero indicates a locked record and any other value locks it.

And that dear friends, covers the basics of file manipulation on the Model 16. In Chapter 10 we shall put these ideas into action.

Chapter 9
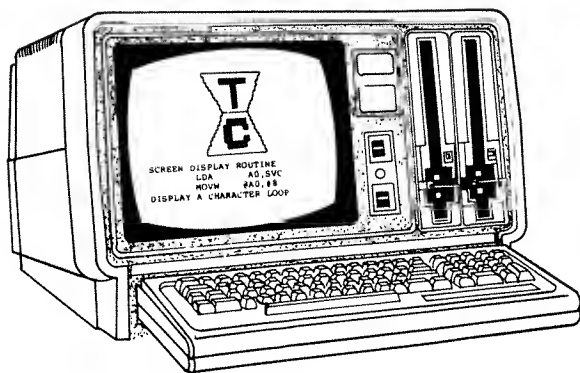
Deciphering Error Codes

The computer can help us debug our programs by detecting such unforgivable sins as syntax errors and illegal addressing mode attempts.

This chapter is divided into two sections. First, we will show how to make TRSDOS 16 inform us of any errors that may occur from executing a supervisor routine; errors that might otherwise go undetected.

Part two will deal with specific error messages that may appear when developing an assembly-language program.

AN ERROR HANDLER

There are times when it is desirable to place error detection lines within a program to check to see if certain conditions are being met.

One application using error detecting is checking to see if everything went according to plan after the execution of a supervisor call. For example, if we call a routine that writes a record to the disk, we would need to know if that record was properly written or if the computer had trouble carrying out our orders. It is possible, for instance, for the disk to be full; the machine would then be unable to store the last entry. Perhaps the record could not be written because there was no such file name on the disk.

The Status Register

Before we proceed any further, we need to discuss one of the MC68000's registers; the *status register*. This is analogous to the status or "flag" registers found in other microprocessors. We do not use this register in the same way we use the address and data registers which we have been working with so far. The status register is comprised of sixteen bits, for a total of two bytes. Eight bits make up one byte. We are only concerned with one byte. The other byte is reserved for use by the machine. Each bit can contain either a one or a zero. Regardless of the type of microprocessor you are working with, the terms "set" and "cleared" are used to describe the state of a bit in the status register. After performing certain operations, the microprocessor "sets" or "clears" some of the bits in this register. This register is also known as a "flag" register.

After executing many op codes such as those that perform arithmetic operations, some of the bits in the status register are set or cleared, giving us an indication of certain conditions which took place as a result of a particular operation.

For example, one bit in the status register is designated a "C" or carry bit. If two numbers are added together by an ADD instruction, the carry bit is cleared when no carry is generated as a result of that addition, and it is set if the operation produces a carry digit.

The MC68000 has many op codes which let us check the status of all eight bits within the register. Then we can take action depending on the condition of the particular bit we are interested in. This testing for a true/false condition is similar to the IF-THEN statement used in BASIC programming.

Displaying an Error Code

With this background information, we are ready to develop a routine that will test for any abnormalities after execution of a supervisor call.

After calling one of the routines contained in the disk operating system, the computer will place an error code in byte-offset two and three in our established SVC block. If TRSDOS was able to carry out our call without any errors occuring, a zero is placed in this position within the SVC block.

By using the op code TEST, the operand, which is byte-offset two in this case, is compared to the value zero. If both are zero, then there is no error and the "Z" bit position (stands for zero bit) of the

status register is cleared. Should there be an error code present, the Z flag is set.

Next, the "branch on condition" instruction is used to check the status of the Z flag. The op code BNE represents "branch if not equal." If the Z flag is set, that indicates there is an error code present. Actually the computer compares the Z bit with the number zero and acts on the results. If the Z flag is set, that is, contains a logical value of one, the result would be "not equal to" and execution would branch to the address location indicated in the operand. Otherwise the program chugs along its merry way onto the next instruction.

To make life as easy as possible, we will define the address to branch to if an error occurs as ERROR HANDLER. This label is defined later in this program.

```
TESTW      2@A0        *ASSUME A0 CONTAINS ADDRESS OF SVC BLOCK
BNE        ERROR HANDLER   *BRANCH IF THERE IS AN ERROR
```

In summary, TEST sets or clears the Z flag, while BNE checks the status of that bit and jumps to another section within the program if the conditions are met.

Now that we have detected an error and sent the program to the section labeled ERROR HANDLER, we need to call up the TRSDOS routine which will jump out of our program and display the associated error code on the screen.

The error handler listing appears below and is followed by an explanation.

```
ERROR HANDLER    LDW     .A1,2@A0
                 LDA     .A0,SVC BLOCK
                 MOVW    @A0,#39
                 STW     .A1,6@A0
                 BRK     #0
```

As we have mentioned, if an error has occurred during the execution of a supervisor call, the TRSDOS error code is placed in byte-offset two of the SVC block. The first instruction,   LDW .A1,2@A0   loads that number into register A1.

Next, we prepare to execute another supervisor call. This built-in routine takes the error code number stored in byte-offset six and jumps to TRSDOS to display the word ERROR and the associated error code number. So as with all of our supervisor calls, we load the address of the block into A0.

The TRSDOS 16 error display routine is number 39, which is placed in byte-offset zero.

Instruction   STW A1,6@A0   takes the error code previously loaded into A1 and stores it into byte-offset six, the position in which we place the code number that we want displayed.

The   BRK #0   instruction causes the error to be displayed.

You may recall the discussion in Chapter 26 regarding why we must first load A1 with the error number and then load A1 back into the SVC block. This is a way of loading a byte-offset into a byte-offset. If necessary, refer to the section on addressing modes in Chapter 2 for a more complete explanation.

It is possible that you do not wish to return to TRSDOS 16 to display the error but would rather have the computer act upon that information. This could be done also. If a "disk full" error occurred you may want the program to create a new file on another disk drive and continue on. There are endless possibilities for handling errors, but all of these can be built around this fundamental basic error-detecting routine.

Displaying an Error Message

Definitions for error codes are listed in the Model 16 owner's manual. But it is not always convenient to get out that big book and thumb through it to find out what a particular error code represents.

The disk operating system contains a routine which, when called, will display a more detailed message concerning the specified error code number.

By placing the number of the error code into byte-offset six and seven, we can call this subroutine and get a descriptive message. We would then get more information than we would by using the previous routine which only displayed the TRSDOS error code number.

The message must be placed somewhere in memory by the display an error message routine. This is established prior to calling the routine. Once it is there, we can jump to the display a line routine and print out the information. Descriptions are 80 characters or less in length. Therefore, the storage area in which we will place the error message should be 80 bytes of space.

The identifying function code is 52. The basic routine follows.

```
LDA     .A0,SVC BLOCK
MOVW    @A0,#52
MOVW    6@A0,#ERROR CODE NUMBER
MOVL    8@A0,#ERROR DESCRIPTION
BRK     #0
```

```
SVC BLOCK
          RDATAB   32,0
ERROR CODE NUMBER
          EQUW     0    *(REPLACE ZERO WITH DESIRED CODE)
ERROR DESCRIPTION
          RDATAB   80,0
```

A sample listing appears below showing how you might use this routine in conjunction with the display a line supervisor call. It is assumed that a supervisor call has just been executed prior to this routine. If an error occurred, it is placed in byte-offset two and three of the SVC block.

```
*GET ERROR FROM SVC BLOCK (ASSUMES ROUTINE HAS JUST BEEN
*CALLED)
          LDW      .A1,2@A0    *GET ERROR CODE-PUT IN A1
          LDA      .A0,SVC BLOCK
          MOVW     @A0,#52     *FUNCTION CODE-ERROR MESSAGE
          STW      .A1,6@A0    *PUT ERROR CODE IN OFFSET 6 & 7
          MOVL     8@A0,#ERROR DESCRIPTION
          BRK      #0          *STORE MESSAGE IN "ERROR DESCR."
*PLACE DESCRIPTION OF ERROR ON VIDEO
          MOVW     @A0,#9      *FUNCTION CODE-DISPLAY A LINE
          MOVW     6@A0,#80    *LENGTH OF LINE TO BE DISPLAYED
          MOVW     8@A0,#13    *TERMINATOR CHARACTER
          MOVL     10@A0,#ERROR DESCRIPTION
          BRK      #0          *DISPLAY ERROR MESSAGE ON TUBE
```
. . . . program continues here . . . .
```
          SVC BLOCK
                   RDATAB   32,0
          ERROR DESCRIPTION
                   RDATAB   80,0
                   END
```

First the routine gets the error code from the SVC block and stores it in offset six and seven in preparation for calling the display an error message routine. A spot in memory is chosen to store the 80-byte long descriptive error message. It is designated here by the label ERROR DESCRIPTION. The supervisor subroutine is ready to be called.

Next, the text now stored in "ERROR DESCRIPTION" needs to be directed to the video display. The function code is loaded into its proper position in the SVC block. The length of the line, 80 bytes, is placed in offset six and seven. We used an ASCII code of 13 as the terminator character.

Finally, the address of the text to be displayed is placed in offset 10, 11, 12, and 13, namely ERROR DESCRIPTION.

ASSEMBLER ERROR MESSAGES

The assembler is capable of displaying many error messages.

Here we will examine some of the more common error messages that you are apt to get when entering the routines presented in this book. Use this section as a reference to help you figure out what the problem might be that causes a specific error message to occur in your programs.

We have classified these errors into two categories, those that occur during assembling and those which show up during the actual running of the program. The linker program is also capable of detecting errors. However, in our experience so far, we have never had an error message displayed during the linking process.

Operand Not Compatible with Instruction

When the operand is not in a form that is allowed for use with the particular op code, an OPERAND NOT COMPATIBLE message is generated. This can happen when an illegal addressing mode has been used.

Let's look at a sample line. In Chapter 2 various addressing modes were shown. As stated there, the MOVe instruction must be used when loading an address register indirectly with a number. If you attempt to use a load (LD) instead, this error would occur. The following line would cause such an assembler error.

```
LDW      @A0,#255
```

Missing Comment Separator and Incorrect Number of Operands

If you get a lot of errors during an assembly, don't be too disheartened. Many times one mistake will cause more than one error message, so you may not have as many problems as first appears. For example, take the instruction

```
STL      .A3,/STORE
```

which takes the current number stored in address register A3 and stores it in the program in the area with the identifying label STORE. If the comma is mistakenly omitted, three error messages will be generated: MISSING COMMENT SEPARATOR, INCORRECT NUMBER OF OPERANDS, and OPERAND INCOMPATIBLE WITH INSTRUCTION. Therefore, by repairing that one problem, three errors will be eliminated.

Unknown Op Code and Illegal Statement

When writing programs in BASIC on most microcomputers,

we get the message SYNTAX ERROR when we have used an instruction that the computer's BASIC interpreter does not have in its vocabulary. The machine does not understand the command, and thus it does not know how to handle it. When writing in assembly language, we have an additional factor that we must consider, and that is the proper column placement of instructions. As covered in Chapter 2, each line is divided into four columns titled LABEL, OP CODE, OPERAND, and COMMENT. We must be sure to place the correct item in its true position. If we placed the MOVe OP CODE in the LABEL or left-most column and accidently placed the operand in the OP CODE column as might happen if we forget to use the TAB key on the Model 16 prior to entering the line, we would get UNKNOWN OP CODE and ILLEGAL STATEMENT error messages. In the following example, MOVW and 10 A0,#MES-SAGE   are in the wrong columns.

```
MOVW         10@A0,#MESSAGE1     *LOADS ADDRESS OF MESSAGE1
SVC BLOCK
             RDATAB     32,0
```

Symbol Undefined and Illegal Expression

The editor/assembler programs which are supplied with the Model 16 are fairly liberal when it comes to the exact positioning of items in the four columns. As we pointed out under UNKNOWN OP CODE just a minute ago, the characters placed at the far left are considered to be labels. After that, the assembler simply requires a minimum of two spaces between each element on a line to separate them. In the following sample listing, you will note that there is only one space between the operand   10@A0,#MESSAGE   and the comment

```
*LOADS ADDRESS OF MESSAGE.
        MOVW     10@A0,#MESSAGE1 *LOADS ADDRESS OF MESSAGE
```

The assembler will display the following error information:

```
        SYMBOL UNDEFINED: LOADSADDRESSOFMESSAGE
        ILLEGAL EXPRESSION
```

A SYMBOL UNDEFINED error will also occur if a label was not defined or was mistakenly defined after the END instruction. Apparently the assembler does not look past the END command, and indeed it is a good thing it does not, since this helps trap errors for us. If a SYMBOL UNDEFINED error has been generated by placing a label beyond an END pseudo op, you can quickly find out

65

by checking the bottom of the assembled listing since every instruction which follows an END will also be surrounded by the error message: STATEMENT ILLEGAL AFTER END.

Symbol Multiply Defined

When you have used a label more than once in a program, a SYMBOL MULTIPLY DEFINED message is produced. The error will show the name of the offending label. This message will appear at every occurrence of the label in the program. Therefore, if you have used a label twice to define two different address locations, you will get two error messages. The sample listing below shows how the error message SYMBOL MULTIPLY DEFINED JUMPDOS will cause two errors when assembled. JUMPDOS appears twice in the listing.

```
JUMPDOS       MOVW       @A0,#264
              BRK        #0
ERROR ROUTINE
              LDW        .A1,2@A0
              LDA        .A0,SVC BLOCK
              MOVW       @A0,#39
              STW        .A1,6@A0
              BRK        #0
              TESTW      2@A0
              BNE        JUMPDOS
JUMPDOS       MOVW       @A0,#264
              BRK        #0
                :
                :
```

Value Not Relative to Current Psect.

If this error occurs, check to see that you have remembered to include pound signs (#) in the operands that require them. As you will recall, the pound sign tells the computer that the numerical information is immediately following. This error would happen if the pound sign was left out of the line or in

```
MOVW       @A0,264
```

The correct format would of course be

```
MOVW       @A0,#264
```

Assembler Warnings

At the completion of assembling a program, the assembler gives a summary report.

```
STATISTICS OF THIS ASSEMBLY

TOTAL NUMBER OF ERRORS     0

TOTAL NUMBER OF WARNINGS   0
```

If there are any errors, the program would not run if execution was attempted, so it would be of no use to proceed with the linking operation.

However, a warning does not necessarily mean a fatal error exists. It is possible to get a warning and yet still have the program perform flawlessly.

A common cause for a warning message is because the assembler has made an assumption. You do not know whether the assumption it made was correct or not, thus we get the cautionary report. If, for instance, you used a load (LD) instruction and did not include a data size suffix (B, W, or L), a warning would be generated.

As we have said, the assembler does automatically make some assumptions. If a size code is not indicated by the programmer, the assembler will evaluate the op code and provide its own code. This will cause a warning message to be displayed after assembling. If the assembler has made a correct assumption as to the data size, the program will run fine, assuming of course everything else in the listing is kosher.

Let us assume a program was written containing the following line:

```
ADD      .A3,#H'5
```

The programmer forgot to include the data size code. This causes the assembler to give the warning message WARN-ING   WORD LENGTH ASSUMED.

In this case, the assembler happens to correctly assume that the instruction should be ADDW. So this line will run fine, but it is advisable to include the suffix when you write your programs. This way you will always be sure exactly how the computer is going to handle your instruction.

Warning 1 Missing String Terminator

Such a message is generated when one of the quotes or apostrophes have been left out of a quoted string of characters as, for example, in TEXT '<A>DD A NAME. Note the end apostrophe has been accidentally left out.

EXECUTION ERROR MESSAGES

It is possible to write a program that will not give any errors or warning messages during the assembling and linking processes, and yet still have errors in it. The program will run fine until a problem is encountered, at which time execution stops and an error message is output to the screen.

Illegal Instruction Trap

When the computer attempts to execute code which it cannot, an ILLEGAL INSTRUCTION TRAP AT USER PC=(relative address number) message is generated. Naturally the assembler would pick out any illegal code during the assembling process so when this error occurs you know that your syntax is not the offending cause.

Check your program to see if your computer is trying to execute a table, or perhaps an area of memory that has been defined as a storage area, such as would be created by the TEXT op code. If you place an SVC block or define some labels at the end of a program, be sure that the computer does not try to execute those lines: it might try to interpret them as instructions. Usually a program will have a call to return to the disk operating ready mode which would be located ahead of any blocks or tables. Should your program contain such lines at its beginning, be sure that you instruct the machine to begin execution after they appear in the program. If you place tables in the middle, be certain to direct the program flow around them.

The following program, if attempted to be run as it stands, would generate such an error.

```
START      LDA      .A0,SVC BLOCK
           MOVW     @A0,#8
           MOVW     6@A0,#30
           BRK      #0
SVC BLOCK
           RDATAB   32,0
           END      START
```

After the machine has carried out the BRK instruction, it proceeds on to the SVC block where it will try to execute the code which has to be generated by the defining of the SVC block. This will create an error code. Refer to Chapter 4 for more information.

Odd Address Trap

Most of the op codes for the MC68000 microprocessor are two

bytes in length. It is important that the operand of certain instructions fall on even address locations in memory. This is a factor which never needs to be considered when programming 8-bit processors.

The error message ODD ADDRESS TRAP AT USER PC=(relative address number) occurs if this uneven memory location problem exists in your program. The relative address number listed in the error message gives us the relative location within our program. This is not an absolute memory address in RAM.

The odd address problem can occur if you define a byte on a line which preceeds an instruction. For this reason it is always a good idea to define a word, even though a byte would suffice.

If you get an odd address trap message and you have some text to be displayed within your program, the problem may be that you have an uneven number of characters in a text string. Normally it would not make any difference whether there was an even or odd amount of letters, but if there are instructions to be carried out which follow an uneven amount of text, then these instructions may fall on an uneven address. Count the characters in your text messages and see if this is the case.

Message Text 'Inventory Program'

If indeed this is the cause, it can be solved in one of two ways. You could increase the length of the text message. The easiest way to accomplish that would be to add an extra blank space (from the space bar) into the message, perhaps at its end. This, of course, is not always possible, since other information may be needed to be placed at that same location on the screen.

Another way is to place all text messages at the end of the program, being careful not to place any instructions beyond these message definitions, with the exception of the END pseudo op. If an uneven amount does give the END instruction a problem, merely define a byte of memory prior to it using DATAB 0. Perhaps an extra byte could be set aside following any line of text in the program which is uneven. That may keep uneven text messages clearly marked. The computer certainly has memory enough so that wasting a few bytes here or there won't matter.

Chapter 10
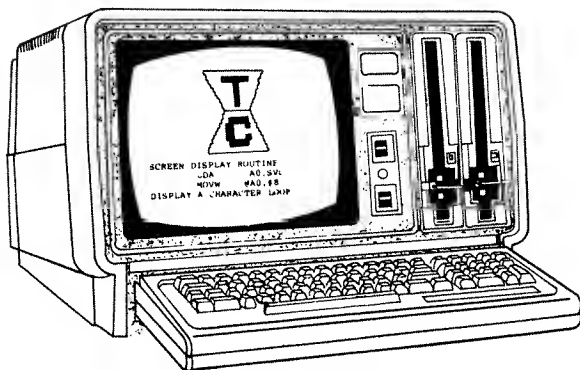
Putting It All Together

The routines developed in previous chapters form the basic ingredients for any type of program you may need to write. Here we will expand on those short listings and bring them together to create a complete disk-based mailing list program. The structure of this program is such that with only minor changes it could be rewritten to handle any type of simple disk storage application. This might include a program that keeps a record of bank account checks or perhaps a program that stores a list of inventory items.

MODULAR FORMAT

Since we are now going to have a much more complex program, many of the short routines, or modules that we have been developing should be used as subroutines to keep the program from becoming unnecessarily long. We can then jump to any subroutine we need by using the CALL and RET statements. We will need to access such routines as clear the screen, jump to DOS, and display a character from many different locations within the program.

For that reason, one of the first things to do is to create a subroutine section and place the various short routines together.

The more lengthy a program becomes the more important it is to keep the listing clear and uncluttered. A complicated program can be kept manageable by the use of comment lines to explain the reasoning behind a particular instruction and to set aside each module or subroutine. As previously mentioned, comment lines,

which begin with an asterisk (∗) symbol, are not present in the assembled object code. Therefore they do not take up any memory space in the final program, nor do they affect the speed of its execution. By examining a paper printout created by assembling a program which contains a comment line, you will note that no machine code is generated nor is any memory space allocated for it. So the mailing list program we will develop here will use comments liberally.

THE GAME PLAN

First we should decide what functions we want the program to perform. Since all actions must branch from the same beginning point, we might draw up a "menu," that is, a list of options from which the operator may choose. The menu will be our home base of operations. Whenever we have completed a job, such as adding a new entry to the list or printing labels, we will return to this point.

In our mailing list program we will need to be able to add a new name and address to the list, modify or make changes to an existing name and address, search for a specific name to see if it is in our file, and print mailing labels. When we are done with the program, we want to be able to "bail out" and return to the TRSDOS ready mode.

With that in mind, we will set up the first display screen to look like this:

```
                             MENU
------------------------------------------------------------


    <A>   Add a name to the list

    <S>   Search for a name

    <M>   Modify a record

    <P>   Printout address labels

    <R>   Return to DOS Ready


Select the appropriate letter >  .
```

Many programmers use a "greater than" and "less than" sign to surround letters in the menu indicating that those are valid characters with which he should respond. In our menu here, we prompt for an "A," "S," "M," "P," or "R" keyboard response.

INITIAL SETUP OF SUBROUTINES AND STORAGE AREAS

The bottom section of our program will contain all the necessary subroutines and storage areas we will require. Over the years microcomputer assembly-language programmers have come to use a line of asterisk symbols in their source code listings to give a clear visual indication of the different sections in their programs. The asterisk also happens to be an Assembler-16 convention instructing the computer that the following information is a comment or documentation for the programmer only and the computer is to ignore it. So a string of asterisks, like the following, need not be preceded by any other symbol.

```
*********************************************
*               SUBROUTINES                 *
*********************************************
```

Assembly-language programs can get very complex and we cannot stress enough how important it is to freely section off each part of the program that performs a specific function.

JUMP TO DOS SUBROUTINE

Let's start our subroutine section by placing several routines in it that we will need to call from the first few lines of our program, namely "clear the screen" and "display a character" instructions. The menu also gives us the option to return to the TRSDOS ready mode so perhaps we should place a routine to jump to DOS here. Note that even though JUMPDOS is a subroutine, we do not need to terminate it with a RET statement since the very act of calling the routine sends it out of the program.

```
*****************************************
*       JUMP TO DOS READY MODE          *
*       ENTRY CONDITIONS: NONE          *
*****************************************
JUMPDOS

        MOVW    @A0,#264   *FUNCTION CODE

        BRK     #0
```

Note the comment in the heading "ENTRY CONDITIONS: NONE." Perhaps it is a good idea if we place this line at the beginning of each subroutine to show any registers or storage areas that must be set up prior to entering the subroutine. In the section

that displays text on the screen which we shall discuss in a moment, that very thing comes into play.

CLEAR THE SCREEN SUBROUTINE

In Chapter 4 we showed a routine that clears the screen by using the display a character supervisor call and sending an ASCII control code to the video display. Again we will call on that routine to perform the duty of clearing the screen. This routine is definitely needed as a subroutine since we will want to call it every time we enter a new phase of our program; at the beginning, when entering a name, changing a name, searching for a party, and the like.

```
***********************************************
* CLEAR THE SCREEN: ENTRY CONDITIONS: NONE *
***********************************************
CLEAR SCREEN
          LDA         .A0,SVC BLOCK
          MOVW        @A0,#8  *FUNCTION CODE
          MOVW        6@A0,#H'1B  *ASCII CODE TO CLS
          BRK         #0
          RET
```

We must remember to place a RET instruction after each routine to send execution back to where it came from.

SCREEN DISPLAY SUBROUTINE

There are almost always several ways in which the same task can be accomplished. In order to display all of the instructions for our menu, we need a routine that will direct text to the video screen. Again, there are many ways to do it but we have opted for a method which uses the "display a character" supervisor call. Is there a method to our madness? We voted against using the "display a line" routine because it requires that we know the length of the line to be displayed. Since we intend to have many lines of text, we would have to devise some way of getting this information into the SVC block. It can be done but we felt it would be easier to use the "display a character" routine and print one letter at a time. In this way we can also test each character. By doing it this way we can also test certain characters or codes in the text line to be used for whatever purpose we wish. We do not need to know the length of the line and we can mix carriage return and line feed codes into the message areas.

In Chapter 6 we developed a routine for printing messages on the screen based on the "print a character" supervisor call. The same concept is used here.

The only thing we need to tell the computer going into this routine is the address location of the first character to be printed. Keeping in mind that this section of the program is to be a subroutine, one that must be compatible with other sections of the program that may call it. For that reason we cannot directly load the address where our menu text is to begin, since when we get to the "add a name to the list" section we will need the address location indicating the beginning of the text to create that particular screen display. So we must set up a storage area where we can place the address pointing to the first character to be printed and fill that area with the location of our choice prior to entering the routine. A moment ago we discussed that very idea. In our comment line heading for this section of the program, we will indicate that a storage area, which we have designated with the label STORE1, must point to the address where the next line to be displayed starts. Now the routine is universal and can display any text that we have stored at any location.

The routine is shown below followed by an examination of what takes place.

```
**********************************************
*    SCREEN ROUTINE ENTRY CONDITIONS:        *
*STORE1 POINTS TO ADDR. NEXT LINE TO DISPL*
**********************************************
SCREEN DISPLAY ROUTINE
        LDA     .A0,SVC BLOCK
        MOVW    @A0,#8  *FUNCTION CODE
DISPLAY A CHARACTER LOOP
        LDL     .A3,/STORE1
        MOVB    7@A0,@A3
        MOVB    6@A0,#0    *INSURES A ZERO IS THERE
        ADDW    .A3,#1    *INCREMENTS
        STL     .A3,/STORE1  *STORES AWAY ADDR NEXT CHAR
        BRK     #0
        LDL     .D0,#0
        LDB     .D0,7@A0
        CMPB    .D0,#5
        BNE     DISPLAY A CHARACTER LOOP
        RET
```

By creating a loop each character can be displayed one at a time. As we have mentioned, the entry conditions require that the address of the text to be printed be stored in STORE1. That is accomplished by the instructions

```
        LDA     .A3,MENU
        STL     .A3,/STORE1
```

Those instructions will appear at the top of our program just prior to calling this subroutine.

Upon entering the screen display subroutine, we first load register A0 with the address of the SVC block. Next the TRSDOS identifying function code, eight, is moved into byte-offset zero. Now we enter a loop which will get a character from memory, print it, and test to see if it has reached the end of the text to be printed. We use an ASCII code of five to indicate the end of the text. As was explained in Chapter 6, we selected this value because it is one of the codes not used by the machine. If you ever write a routine where you choose a value that is used by the machine to cause an action on the screen, you may need to slightly revise our "display a character loop" section so that the test for your terminator character is done before the character is sent to the video. Here we send the character to the screen and then test to see if it is the terminator character. It doesn't make any difference here which way it is handled since sending an ASCII five to the screen does not do anything anyway.

Inside the "display a character" loop we load the address location of the first byte of our text message into register A3. We could have chosen any address register at this point for this instruction with the exception of A0 since that is keeping track of the SVC block.

```
LDL      .A3,/STORE1
```

The slash indicates "the contents of." So the contents of STORE1 is loaded into register A3. Addresses require four bytes to store them so the load instruction must indicate a long word.

Byte-offset seven holds the value of the ASCII code to be sent to the screen. Most if not all of the instructions for the MC68000 are a word in length so we insure that there is no stray value in offset six by placing a zero into it.

At this time, register A3 holds the address of the character we want to be printed. After that character is displayed, we will need to get the next one. All of the text or control codes for a given display will have to be placed in sequential order. In this way we can simply increment the address stored in A3 by one and step through each character.

```
ADDW     .A3,#1      *INCREMENT
```

Once we have incremented the address to point to the location of the next character to be sent to the supervisor routine, that address must be stored away into STORE1, our storage area. It will be recalled and used to print another character if the present character does not terminate the routine.

Finally, we test the character to see if it is our terminator code, five. The CMP instruction enables us to compare a numerical value with the value stored in a data register. To insure there are no stray numbers in the data register we select, zeros are loaded into it. Register D0 was selected but any data register would do equally well. Then the one-byte character that was just displayed on the screen is loaded into D0 where it is compared with the number five. If there is a match then the routine has, hopefully, successfully completed the entire screen display. The RETurn instruction ends the subroutine. Should there not be a match, execution branches back to the "display a character loop" location where another character is obtained, placed into the SVC block, and displayed.

The text which will comprise our menu needs to be defined. Following the subroutine section of our program we have set up a message and working storage area (we've adopted this phrase from the COBOL language). In some ways assembly-language programming on the Model 16 resembles certain aspects of COBOL—for instance the use of whole phrases to define a location in the program. In any event, we can use the label MENU to define the first address location where we will place the various text and control codes that will make up the full screen display. Since the display a character routine permits us to mix text and codes, we can use the op code TEXT to define the alphanumeric characters we wish to print and DATA to send line feed/carriage return instructions. If we need a line feed we can use DATAB 13. Should we want several line feeds, DATAB 13, 13 will work.

The first thing we want to print is the word "MENU" and underline it with a row of 80 equal signs (=) to dress up the screen display. The editor program does not let us assign a string of 80 characters on one TEXT line. This problem is easily solved by simply placing 40 equal signs on one line and 40 on another without sending an ASCII code 13 to give a line feed. A continuous row of 80 symbols can then be displayed.

```
MENU    TEXT    'MENU'
        DATAB   13,13
        TEXT    '========================================'
        TEXT    '========================================'
        DATAB   13,13,13,13
        TEXT    '<A>  Add a name to the list '
        DATAB   13,13
        TEXT    '<S>  Search for a name'
        DATAB   13,13
        TEXT    '<M>  Modify a record'
        DATAB   13,13
        TEXT    '<P>  Printout address labels'
```

```
DATAB    13,13
TEXT     '<R>  Return to DOS Ready'
DATAB    13,13
TEXT     'Select the appropriate letter  >
DATAB    5          *TERMINATOR CHARACTER
```

Note that we have added a space at the end of the message "<A> Add a name to the list." This is to make the message an even number of bytes in length in order to prevent a possible odd address trap error later on in the program.

While we are adding information to our storage area, we can establish the SVC block and STORE1. STORE1 will be used to store an address so it will have to be four bytes wide.

```
SVC  BLOCK
            RDATAB   32,0
STORE1   RDATAB   4,0
```

Now to jump back to our subroutine section and add one more routine. We are skipping around a bit but we feel that it is better to explain the entire program in the order that you might write it instead of starting at line one and explaining each line in numerical order. By doing it this way, it is easier to see the relationship between the various sections of the program and how they tie together. The thinking is more logical.

There is one more subroutine we want to put into place before going back to the top of the program and continuing on. After executing some of the supervisor calls, we may need to test the SVC block to see if any errors have occurred during the carrying out of the routine.

All supervisor calls return an error number in byte-offset two and three of the SVC block after execution. The error handling routine found in Chapter 9 should be placed in the subroutine section of our mailing list program. There will be a time when we want to check if an error is there but not print it on the screen. An error in the open routine will need to be handled differently as we shall see when we get to the file handling section.

```
*****************************************
*      DISPLAY ERROR NUMBER ON VIDEO      *
*****************************************
ERROR HANDLER
          LDW    .A1,2@A0        *LOAD ERROR CODE INTO A1
          MOVW   @A0,#39         *IDENTIFYING FUNCTION CODE
          STW    .A1,6@A0        *LOAD ERROR CODE INTO SVC
          BRK    #0              *EXECUTE ERROR NUMBER DSPLY
          RET
```

THE MAIN BODY OF THE PROGRAM

The top of our program is indicated by the label START. Whenever we need to return to the menu we can branch to the START label.

As usual, the screen should be wiped off before we establish our menu page. Our first instruction initiates a jump to the clear screen subroutine which we already have in place:

```
CALL    CLEAR SCREEN
```

We have placed the text we wish to use in the working storage area. To add some "class" to the menu display page, let's horizontally center the word "MENU" at the top of the screen. This is done by positioning the cursor at the location where printing is to begin and then calling the routine to display the desired information. In Chapter 4 the supervisor call that positions the cursor was discussed. In this case we want the column position to have a value of one, indicating the top line on the screen. To calculate the horizontal location, we use standard typing methods. There are 80 character positions across a horizontal line. Half of that is 40, about the horizontal center point. MENU has a length of four characters, half of which yields two. The result of 40 minus 2 is 38, the location where printing should begin.

```
POSITION CURSOR
        MOVW    @A0,#10     *POSITION CURSOR FUNCTION CODE
        MOVW    6@A0,#1     *ROW POSITION (TOP TO BOTTOM)
        MOVW    8@A0,#38    *COLUMN POSITION (LEFT TO RIGHT)
        BRK     #0
```

Since register A0 still holds the address of the SVC block which was loaded in from the previous routine, the instruction LDA .A0,SVC BLOCK is not needed in the position cursor section.

We have cleared the screen and positioned the cursor. Now we can put the screen display ROUTINE to work. As we said before, the address where the first character in our text is stored must be loaded into STORE1. This is the entry condition for the screen display ROUTINE. To do this, we randomly chose A3 to temporarily hold the address of the menu. Then that number is stored away into STORE1. The screen display routine can be called next. The entire video screen will show our menu and prompt the user to make a selection.

```
INITIAL SETUP
        LDA          .A3,MENU
        STL          .A3,/STORE1    *STORES AWAY BEGIN. ADDR.
        CALL         SCREEN DISPLAY ROUTINE
```

The operator has been shown a list of options to choose from. We can then call on the keyboard supervisor routine to get the character he types and place it into a holding area where we can work with it. Chapter 6 shows the routine in which we place the number of characters to be entered and the memory location where the input is to be stored into the SVC block. We want the routine to wait until the operator hits one key. It is not necessary that he hit the <ENTER> key. We do not want him to enter more than one letter. Also it will speed the operator's use of the program if he only has to hit one key as opposed to also hitting <ENTER>.

In this instance, only one byte needs to be set aside to store the character entered from the keyboard. The holding place is defined down in the working storage section of our program.

```
                KEYBOARD STORE1
                        DATAB       0́
                        DATAB       0
```

To prevent a possible odd address trap error, we define two bytes, even though only one is needed for this routine. This keeps address locations on even numbers.

```
KEYBOARD INPUT ROUTINE
        MOVW         @A0,#5       *FUNCTION CODE
        MOVW         6@A0,#1      *NUMBER OF CHAR. TO BE ENTERED
        MOVL         8@A0,#KEYBOARD STORE1
        BRK          #0
```

So far we have displayed the menu and gotten an input from the user. Next we must test that character to find out what he entered. This can be accomplished by loading that character into a data register. Once it is in a register, the CoMPare and Branch instructions can test it and branch execution to the appropriate section of the program.

To load the character into a register, we use the instruction:

<p align="center">LDB   .D0,/KEYBOARD STORE1</p>

A byte suffix (B) is attached to the LoaD command as that is all we need to check. We arbitrarily chose data register D0. Remember that the slash symbol (/) is an Assembler-16 convention meaning "the contents of," so the contents of the address referred to as KEYBOARD STORE1 is loaded into register D0. We tacked a 1

onto the label since it may make it easier for you to modify this program to make it perform a different task. If at some time in the future you want to take the basic structure of this program and create an inventory storage program or some such thing, you may need other areas within the program to save inputs from the keyboard. By using the label KEYBOARD STORE1 the door is open for the addition of more areas with the same general name: KEYBOARD STORE2, KEYBOARD STORE3, etc.

Now to test the letter entered by the operator. Since there are five options, five similar instruction sequences are needed. The character is compared with the known ASCII value of the letter being asked for. If a match is found, the proper flag in the status register is set. A "branch if equal" (BE) command can direct the program flow to the proper location where execution is to continue. We need to test for the following letters and branch to sections which we will give the corresponding labels to.

```
test for "A"    ASCII code = 65    branch to ADD ROUTINE
test for "S"    ASCII code = 83    branch to SEARCH ROUTINE
test for "M"    ASCII code = 77    branch to MODIFY ROUTINE
test for "P"    ASCII code = 80    branch to PRINTOUT ADDRESS
                                           LABELS
test for "R"    ASCII code = 82    branch to JUMPDOS
```

It is important to "idiot proof" the program as much as possible. We are not implying that the operator may be an idiot (of course, he might be); but as you know it is easy to make a typographical error or to become pretty bleary-eyed after entering a lot of names and slip and hit the wrong key. For this reason we should end this routine by placing a BRanch instruction which will clear the screen, display the menu, and prompt for the correct letter again. This is a blanket instruction in that it will handle any key the user can find on the keyboard. Only the correct responses will cause an action to be taken.

So, the test routine could be put together in a very simple form to look something like this:

```
*TEST INPUT
      LDA      .D0,/KEYBOARD STORE1
      CMPB     .D0,#65    *TEST FOR AN "A" INPUT
      BE       ADD ROUTINE
      CMPB     .D0,#83    *TEST FOR AN "S" INPUT
      BE       SEARCH ROUTINE
      CMPB     .D0,#77    *TEST FOR AN "M" INPUT
      BE       MODIFY ROUTINE
      CMPB     .D0,#80    *TEST FOR A "P" INPUT
      BE       PRINTOUT ADDRESS LABELS
      CMPB     .D0,#82    *TEST FOR AN "R" INPUT
```

```
       BE        JUMPDOS
       BR        START
```

THE OPEN ROUTINE

Since we will need to open the file containing the list of names from many different places in the program, the best way to handle an open routine is to place it in the subroutine section of our program. The file will have to be opened to add new names, search for a name, modify an existing record, or print out mailing labels. Creating the open routine will take a little thought. The TRSDOS supervisor call which opens a file requires that a different value be placed in the SVC block to indicate when a file is first created or when it has previously been opened. Therefore we must make two open subroutines, one to be used the first time the file is opened and created, and the second to open an already existing file.

CREATE A FILE

There are many ways to achieve the same results in programming. We will now show three different ways that the "open a file" supervisor block could be set up.

The first method we will examine is somewhat similar to the example given in the Model 16's owner's manual under the "open" supervisor call heading. That programming technique requires the use of the EQUate op code to place the various information into the appropriate locations, as in the following example:

```
INDENTIFYING FUNCTION CODE
           EQUW       40
FILENAME
           TEXT       'LIST/DAT'
           DATAB      13
PARAMETER LIST
           RDATAB     5,0
FIXED OR VARIABLE FILE
           EQUB       'F'
CREATE A FILE
           EQUB       1
ATTRIBUTE
           EQUB       32
OPEN       MOVW       @A0,#IDENTIFYING FUNCTION CODE
           LDA        .A1,FILENAME
           STL        .A1,6@A0
           LDA        .A1,PARAMETER LIST
           MOVB       @A1,#READ OR WRITE ACCESS
           MOVB       1@A1,#LENGTH OF RECORD
           MOVB       2@A1,#FIXED OR VARIABLE FILE
           MOVB       3@A1,#CREATE THE FILE
           MOVB       4@A1,#ATTRIBUTE
           STL        .A1,10@A0
           BRK        #0
```

You may note that when indirectly loading A1, we used the op code MOVB as shown in the owner's manual. However, we were not able to get MOVW to work and discovered MOVB functioned fine. It makes sense that the byte suffix be used, anyway, since only one byte is being loaded.

In a moment we shall go into more detail about the instructions needed to set up the SVC block before a file can be opened.

But first let's look at another way to set up the create-a-file routine. In this case, the decimal values are located within the move or load instruction itself. By doing it this way, the program can be shortened since all of the equate instructions are not needed.

```
OPEN       MOVW     @A0,#40      *IDENTIFYING FUNCTION CODE
           LDA      .A1,FILENAME
           STL      .A1,6@A0
           LDA      .A1,PARAMETER LIST
           MOVB     @A1,#87      *ALLOWS READ OR WRITE TO FILE
           MOVB     1@A1,#90     *RECORD LENGTH = 90 BYTES
           MOVB     2@A1,#70     *ASCII CODE "F" FOR FIXED FILE
           MOVB     3@A1,#1      *CREATE THE FILE ON DISK
           MOVB     4@A1,#32     *USER ATTRIBUTE
           STL      .A1,10@A0    *PUT ADD OF PARA. LIST IN SVC
           BRK      #0
FILENAME
           TEXT     'LIST/DAT'
           DATAB    13
PARAMETER LIST
           RDATAB   5,0
```

Finally we offer the version which we will use in our mailing list program. In the preceding two instances, the parameter block contains information indicating the logical record length, user attribute codes, read/write accessibility, fixed or variable file, and code to open a new file or an existing one. The various data are placed into those codes by first defining a section of consecutive bytes of memory (parameter list). Similar to the way we have been handling the SVC block, register A1 has been chosen to point to the memory location of the parameter list. By using byte-offset addressing, the values we have selected are indirectly moved into the parameter list block. Then the address of that block is stored into the proper location in the SVC block, after which the supervisor call is made.

In our final method, each value is placed directly into the parameter list memory block. We feel this is the simplest and least complicated technique to use in this particular circumstance. It does not matter whether or not the address of the SVC block is loaded into A0, since its address should already be there. By the time our program execution reaches the open section, many other super-

visor routines, such as clearing the screen and getting input from the keyboard, have already been called. We will not use A0 for any other purpose in our program.

Rather than define the parameter list and filling it with zeros, let's place the values directly into that block. So in the working-storage section, we establish the parameter list with the numbers 87 (allows reading or writing to the file), 90 (the record length in bytes), 70 (the ASCII code for "F," fixed file code), 1 (create the file on disk and open it), and 32 (user attribute code).

```
                PARAMETER LIST
                    DATAB    87,90,70,1,32
```

When writing assembly-language programs for the Model 16, there is one thing that should haunt you and forever be on your mind, that is, the need to keep addresses on even numbers. Now, as you examine the parameter list you will note that there are five values which therefore take up five bytes of memory. While there is nothing wrong with that in itself, as you should well know by now, the number five is an odd number (oh horrors!) and could cause an odd address trap later on in the program if there are any instructions following it. It would be a wise decision to tack an extra byte onto the end of that block and thereby fill it out to six digits. The last digit will never actually be used anyway, so we can stick any value there. Perhaps it is safest to use a zero, as we have here:

```
                PARAMETER LIST
                    DATAB    87,90,70,1,32,0
```

Our create a file routine will look like this:

```
*****************************************************
*   OPEN - CREATE A FILE SUBROUTINE                 *
*****************************************************
OPEN    LDA     .A0,SVC BLOCK
        MOVW    @A0,#40      *IDENTIFYING FUNCTION CODE
        LDA     .A1,FILENAME
        STL     .A1,6@A0
        LDA     .A1,PARAMETER LIST
        MOVB    3@A1,#1      *CREATE THE FILE ON DISK
        STL     .A1,10@A0    *PUT ADDR OF PARAM LIST IN SVC
        BRK     #0
```

To be on the safe side, we included the instruction LDA .A0,SVC BLOCK to be sure A0 is pointing to the right place.

Next, the identifying function code, decimal 40, is moved into the SVC block. Then the address of the file name is loaded into SVC

block by first loading in into an address register with a load instruction and then into the SVC block with a store command:

```
LDA     .A1,FILENAME
STL     .A1,6@A0
```

We used register A1 to store the file name into the block. It is not necessary to keep that value in A1 so we are free to use it again. Now we can load the address of the parameter list into A1. As we will see in a moment, the fourth byte in the parameter list indicates whether an existing file is to be opened or a new one created. We can move the number one into the parameter list (indicating create a file) with the   MOVB   3@A1,#1   instruction.

Before we put a RET instruction at the end of our open routine, we need to store away the file identification number. In Chapter 8 we told how TRSDOS creates a number when a file is opened. We must use this number to perform read/write operations as well as close the file, although closing all programmer files can be done by supervisor call number 133, which is a blanket close, but more on that function later.

The file identification number is used by the computer to identify that particular file. We had better store that number away in a safe place—somewhere where we can easily fetch it while executing routines anywhere in the program. The SVC block is used constantly throughout our program. We cannot depend on any values remaining intact in this area. Everytime a supervisor call is made, the block values may change. Let's temporarily use address register A1 to move the file ID number out of the SVC block and into our storage area, which we will give the appropriate label FILE IDENTIFICATION NUMBER.

```
LDW     .A1,14@A0     *GET FILE ID #
STW     .A1,/FILE IDENTIFICATION NUMBER
RET
```

OPENING AN EXISTING FILE

As we mentioned, our program will require two separate open routines, one to open a file the first time to create it on the disk and the other to open an existing file.

There is really only one difference between the routine to create a file or open an existing file, and that is the byte in the fourth position (offset three) in the parameter list. Therefore, this routine will be very similar to the create instructions, the only change being the presence of a zero in the fourth byte of the parameter list.

84

Now, the program is run and the operator selects from the menu the option to add a name to the file, how will we know which open routine to call? If we try to open a file by creating it and the file already exists, the machine will return an error into the usual location in the SVC block, i.e., offset two and three. What we need is a test after the create routine is executed to see if an error occurred. If so, we can branch execution to the existing open file section. The completed create routine will look like this:

```
*****************************************
*  OPEN - CREATE A FILE SUBROUTINE      *
*****************************************
OPEN    LDA     .A0,SVC BLOCK
        MOVW    @A0,#40     *INDENTIFYING FUNCTION CODE
        LDA     .A1,FILENAME
        STL     .A1,6@A0
        LDA     .A1,PARAMETER LIST
        MOVB    3@A1,#1     *CREATE THE FILE ON DISK
        STL     .A1,10@A0   *PUT ADDR.OF PARAM.LIST IN SVC
        BRK     #0
        TESTW   2@A0        *TEST FOR AN ERROR
        BNE     OPEN EXISTING FILE   *IF ERROR, THEN BRANCH
        RET                 *RETURN
```

If an error occurs when the computer tries to create a file, we detect that error and send it to the open existing file subroutine. However, we are taking a chance here in assuming that the error will be a "file already exists" error. If you like, you could put additional testing here to see exactly what type of error has occurred and then act accordingly. But our aim is to demonstrate a disk-based mailing list program in as clear and uncluttered a way as possible. Did you ever get the feeling that a program is never done? There is always something else you could change, always another feature you could add. Theoretically, spend your whole lifetime on a single program.

Now we can set up our open existing file routine. While it may not be necessary, we prefer to try to close the file before opening it. We do this just in case the create subroutine does manage to get the file open. If we then branch to the open existing file and try to open it, a "file already open" error may occur. So again to be on the safe side we will close the file. Nothing can be harmed if the file is already closed, and it only takes a split second to execute the call.

We will, of course, need to close the file from many different areas in our program, so it would be best to make a close subroutine which we can call up from any point. There are two close supervisor calls available to us. One close routine, with a function code of 42, allows one individual file to be closed. This, however, would re-

quire a little extra work from us since we would have to get the file identification code that the disk operating system invents at the time of opening it, and place that into the close SVC block.

Perhaps an easier routine is the blanket close routine, which has a function code of 133. This call closes all user files (not DO or SPOOL files—see the owner's manual for more information on these particular cases). Here all we will need to do is place the identifying supervisor number into the SVC block and call it.

In our subroutine section, we will place the close all files routine.

```
*****************************************
*          CLOSE THE FILE              *
*****************************************
CLOSE
        MOVW    @A0,#133    *CLOSE ALL FILES
        BRK     #0          *CALL TO SVC CLOSE ALL FILES
        RET                 *RETURN
```

The open existing file routine follows.

```
*****************************************
*     OPEN EXISTING FILE SUBROUTINE    *
*****************************************
OPEN EXISTING FILE
        CALL    CLOSE
        LDA     .A0,SVC BLOCK
        MOVW    @A0,#40     *INDENTIFYING FUNCTION CODE
        LDA     .A1,FILENAME
        STL     .A1,6@A0
        LDA     .A1,PARAMETER LIST
        MOVB    3@A1,#0     *OPEN EXISTING FILE CODE
        STL     .A1,10@A0   *PUT ADDR.OF PARAM.LST IN SVC
        BRK     #0
        TESTW   2@A0        *CHECK FOR ERRORS
        BNE     ERROR HANDLER
```

KEEPING TRACK OF NEXT AVAILABLE RECORD

As you know, when the computer is turned off, all of the information in RAM is lost. A way of storing the last record number that contains a name and address must be devised. When we get to the routine which will write information to the disk, you will see how that routine requires that we tell it the record number where that information is to be stored in the file. So we will need a way to find out the last used record so we can write new information into the next higher numbered record. It will also be necessary to know the number of records in the file when we eventually get to the print labels routine. There, each record will have to be read from the disk and directed to the printer. It will be necessary to know what the

highest record number is so we will know when all of the entries have been printed.

Obviously we cannot use a label to establish a location to store the next record number (Yes, that will be needed too). While the program is up and running, we will need a place to store the next available record. But this would not work if we returned to TRSDOS ready or if we turned off the computer and came back at a later date, loaded the program, and attempted to add more names to the list.

Many techniques could be used to overcome this problem. A separate file could be opened and used to just store the top record number in the mail list file.

Perhaps a simpler way would be to dedicate the first record in the mail list file, record number zero, to the job of storing the last record number in the file. Then all we have to do is open the file, which we would need to do anyway in order to do a read or write, and get that information from record zero. This saves a lot of work setting up another whole open routine using a different file name. It won't matter if we waste one record in the file. The machine has such a vast amount of storage, especially if several hard disk drives are attached, that loosing one record is almost like taking one grain of sand away from the beach. No one will ever miss it.

So let us designate the first four bytes of record number zero as the holding tank for the last used record in the mail list file.

The first time we access that file, it will have a zero in it. That can be used to advantage. All we have to do is get the number from record zero and increment it by one. That will give us the next available record number where new data can be written.

We already have a disk buffer area in RAM labeled DISK BUFFER AREA1. The first four bytes (a long word) will store the last used record number.

At the end of our "open existing file" routine can be placed a routine to read the first record on the disk. This information is brought in and stored in the "disk buffer area1". This is accomplished by using the read supervisor call, whose identifying function code is 35 decimal.

The address of the "disk buffer area1 is loaded into address register A1 and then stored into a RAM location which we will call NEXT RECORD NUMBER. Actually this stores the last filled record number in the file. That value is to be used in figuring out the next available record. We must increment the value stored there by one in order to get the next available record number.

Upon entering this routine, the first thing we must do is to store away the file identification number. A moment ago we discussed that under the *open-create a file* section. Initially, program flow will be sent to the "open-create a file" routine. If the file already exists, the error will cause it to jump to the "open existing file" routine. We have not yet stored away the file identification number if the file was already in existence. Since program execution will automatically fall through to the "get last used number" section after reaching the "open existing file" section, we can place instruction to save the file ID number in either the "open existing" or "get last used record" sections. The two instructions can be identical to those that we just developed under "open-create a file."

```
LDW     .A1,14@A0   *GET FILE ID
STW     .A1,/FILE IDENTIFICATION NUMBER
```

With that in mind, we can tack the following routine onto the end of the open existing file section.

```
*GET LAST USED RECORD NUMBER FROM RECORD #0
        LDW     .A1,14@A0,   *GETS FILE ID
        STW     .A1,/FILE IDENTIFICATION NUMBER
        MOVW    @A0,#35      *INDENTIFYING CODE (READ)
        STW     .A1,6@A0     *PUT IN SVC BLOCK
        LDA     .A1,DISK BUFFER AREA1
        STL     .A1,8@A0     *PUT BUFFER ADDRESS IN SVC
        MOVL    12@A0,#0     *GET RECORD #0 (1ST IN FILE)
        MOVW    16@A0,#0     *UNLOCK RECORD
        BRK     #0
        LDL     .A1,/DISK BUFFER AREA1
        STL     .A1,/NEXT RECORD NUMBER
        RET
```

THE ADD ROUTINE

Upon selecting the "add a name" option from our menu, program flow is passed to a section which we will label "ADD ROUTINE." (That sounds like a good descriptive title.)

Once into the add routine, we must open the file, get the file identification number that is created by the open routine, and store that number into the memory location labeled FILE IDENTIFICATION NUMBER. We have already built that feature into our open subroutines.

Next we can clear the video and print a nice heading on the screen, showing the operator that he is in the add mode. We already have a subroutine in place that we can call to print text on the screen. As you will recall, that routine displays one character at a time until the message is complete, and is terminated by a decimal

five. An entry condition must be met prior to calling up the subroutine. The address of the first character to be printed has to be placed in STORE1.

So the first few lines of the add routine can be written something like this:

```
ADD ROUTINE
        CALL    OPEN
ADD MENU
        CALL    CLEAR SCREEN
        LDA     .A3,ADD SCREEN DISPLAY
        STL     .A3,/STORE1
        CALL    SCREEN DISPLAY ROUTINE
```

The name, address, city, state, zip, and special code information has to be entered by the operator, stored within our program, and written to the disk.

The user will have to be prompted with messages to enter the name, the address, and so on. In Chapter 6, we explained the supervisor routine that prints a line of text and gets input from the keyboard. This routine (with an identifying number of 12) requires that the number of characters to be displayed, the number of characters allowed to be entered by the user, the address of the prompting message, and the storage address of the keyboard response, be placed within the SVC block.

In the working storage section of our program, the prompting messages and storage locations for the operator's entries can be established by using labels. Just as we did in the creation of the menu, a blank space is added to the end of each text message that has an uneven number of characters in it.

Once the information has been entered by the operator, that data will have to be written to the disk. By placing all of the input responses in consecutive order, we can give the whole block a label and use it to write the entire 90-byte block to the disk. Note that we determine how many bytes of space are to be dedicated to each parameter. We have chosen to set aside 25 characters for the name, 25 for the street address, 25 for the city or town, 2 for the state, 8 for the zip code, and 5 for the special user code.

```
*PROMPT AND INPUT STORAGE AREA FOR "ADD" ROUTINE
PROMPT1   TEXT   'Enter name > '
PROMPT2   TEXT   'Enter street address > '
PROMPT3   TEXT   'Enter city or town > '
PROMPT4   TEXT   'Enter 2 letter state abbreviation > '
PROMPT5   TEXT   'Enter zip code > '
PROMPT6   TEXT   'Enter code > '
DISK BUFFER AREA1
```

```
INPUT1   RDATAB   25,0    *STORES NAME
INPUT2   RDATAB   25,0    *STORES ADDRESS
INPUT3   RDATAB   25,0    *STORES TOWN
INPUT4   RDATAB   2,0     *STORES STATE
INPUT5   RDATAB   8,0     *STORES ZIP
INPUT6   RDATAB   5,0     *STORES CODE
```

The first time a name entry is added to the list, the entire disk buffer area1, that is, the block of memory reserved for keyboard input, will contain zeros. However, the second time we go for another name the information entered previously by the keyboard will still be there. For example, if the first name entered was longer than the second name, some of the characters of the first name will be trailing on the new name. Therefore, it would be a good idea for us to clear out that buffer area before a new name and address entry is made. Placing zeros in the block should be all right, but sometimes zero is used as a control code and it is best to avoid it. A better choice would be to fill the block with spaces; e.g., an ASCII 32. In this way we are sure that nothing will be seen on the screen or on the printer. Are you thinking ahead? Yes, we can use this same area to dump data to the printer too. But let's not get off on a tangent just yet.

So let's write a small subroutine that will stuff 90 spaces in the disk buffer area1 block. Here is how we chose to do it, although other methods could also be employed.

```
*****************************************
* FILL DISK BUFFER AREA WITH SPACES     *
*****************************************
CLEAR BUFFER
         LDW     .D0,#90    *D0 IS COUNTER - 90 BYTES
         LDA     .A4,DISK BUFFER AREA1
FILL LOOP
         MOVB    @A4,#32    *LOAD SPACE
         ADDW    .A4,#1     *INCREMENT ADDRESS
         DBE     .D0,FILL LOOP    *90 BYTES
         RET
```

Here, register D0 is loaded with a count of 90. The address of the first byte holding a character to be printed is loaded into register A4. Next, we enter a loop where a space is loaded into the address pointed to by A4. The address is incremented by one to point to the next location where a space is to be placed. The value in D0 is decremented. If it has reached zero, then the program execution is returned back to the next statement after the one which called the subroutine. Otherwise, the instructions inside the loop are carried out another time.

Now all we have to do is set up the SVC block for the routine that will print a message and get data from the operator. This could be done in a loop, but for the sake of simplicity we will use the old brute-force method and write the set of instructions six times, once for each needed parameter. The first set is a little longer than the rest because we have placed in it the instruction to load the SVC block with the identifying supervisor number and the instruction to call the "clear the buffer" subroutine.

```
*GET NAME FROM KEYBOARD
          CALL    CLEAR BUFFER
          MOVW    @A0,#12      *FUNCTION CODE-DSPLY TXT-INPUT
          MOVW    6@A0,#14     *# OF CHAR. IN PROMPT MESSAGE
          MOVW    8@A0,#25     *MAX # OF INPUT CHAR. ALLOWED
          MOVL    10@A0,#INPUT1
          MOVL    14@A0,#PROMPT1
          BRK     #0
*GET STREET ADDRESS FROM KEYBOARD
          MOVW    6@A0,#24     *# OF CHAR. IN PROMPT MESSAGE
          MOVW    8@A0,#25     *MAX # OF INPUT CHAR. ALLOWED
          MOVL    10@A0,#INPUT2
          MOVL    14@A0,#PROMPT2
          BRK     #0
*GET TOWN/CITY FROM KEYBOARD
          MOVW    6@A0,#22     *# OF CHAR. IN PROMPT MESSAGE
          MOVW    8@A0,#25     *MAX # OF INPUT CHAR. ALLOWED
          MOVL    10@A0,#INPUT3
          MOVL    14@A0,#PROMPT3
          BRK     #0
*GET STATE FROM KEYBOARD
          MOVW    6@A0,#36     *# OF CHAR. IN PROMPT MESSAGE
          MOVW    8@A0,#2      *MAX # OF INPUT CHAR. ALLOWED
          MOVL    10@A0,#INPUT4
          MOVL    14@A0,#PROMPT4
          BRK     #0
*GET ZIP FROM KEYBOARD
          MOVW    6@A0,#18     *# OF CHAR. IN PROMPT MESSAGE
          MOVW    8@A0,#8      *MAX # OF INPUT CHAR. ALLOWED
          MOVL    10@A0,#INPUT5
          MOVL    14@A0,#PROMPT5
          BRK     #0
*GET SPECIAL CODE FROM KEYBOARD
          MOVW    6@A0,#32     *# OF CHAR. IN PROMPT MESSAGE
          MOVW    8@A0,#5      *MAX # OF INPUT CHAR. ALLOWED
          MOVL    10@A0,#INPUT6
          MOVL    14@A0,#PROMPT6
          BRK     #0
```

WRITING TO THE DISK

At this point, all of the information that is to be stored in a record has been entered by the operator and is residing in the area we affectionately call disk buffer area1. Now the information in that block must be written onto the disk. The supervisor call to write data to the disk is easy to use. The identifying function code, 44, is placed into the SVC block along with the address where the text to

be written is stored, the file identification number that was created by TRSDOS at the time it was opened, and the record number. (That last factor can be a little bit tricky.)

As discussed previously, we will be using the first record in the file to keep track of the last record that contains data. At the time the file is opened, we read that number in the first record and save it in the RAM space labeled NEXT RECORD NUMBER. This value is used to let us know the highest record number in the file. If we add one to that number, we will get the next available record number where a new entry can be written to the disk.

After we write the information to the disk, we must save the value which points to the last record number in the file. Since the SVC block is already set up to perform a write operation, all we need to do is place that number in byte-offset eight and tell the machine that the record to receive this information is record zero. Then a simple BRK instruction will call up the routine.

```
WRITE
        MOVW    @A0,#44     *IDENTIFYING FUNCTION CODE
        LDW     .A1,/FILE IDENTIFICATION NUMBER
        STW     .A1,6@A0    *PUT FILE ID INTO SVC BLOCK
        LDA     .A2,DISK BUFFER AREA1
        STL     .A2,8@A0    *PUT BUFFER ADDR. INTO SVC
        LDL     .A3,/NEXT RECORD NUMBER
        ADDW    .A3,#1      *INCREMENT NEXT RECORD
        STL     .A3,12@A0   *PUT RECORD # TO WRITE INTO SVC
        STL     .A3,/NEXT RECORD NUMBER   *SAVE NEW NUMBER
        BRK     #0          *EXECUTE SUPERVISOR ROUTINE
*SAVE NEXT AVAILABLE RECORD # IN RECORD #0
        LDA     .A2,NEXT RECORD NUMBER
        STL     .A2,8@A0
        MOVL    12@A0,#0     *RECORD #0 TO WRITE
        BRK     #0
```

It is conceivable that the user may want to add many names in one sitting. If we simply returned him to the menu at this point it would require a lot more effort on his part. He would then have to again select the "add a name" option and the file would have to be reopened and the first record read to calculate the next record number.

So to make our program more "user friendly" (that term is getting worn out by the micro industry, but at least it's descriptive) we will ask the operator if he wants to add another name or return to the menu. The file is already open and we have updated the next record number in both RAM and on the disk.

Again we can call on the routine that prints a message on the screen and waits for an input. We only need one byte to store the

response away in memory. A few extra line feeds should be given prior to printing a new message to make it stand out from the other text on the screen. Also, note that we have defined two bytes of memory for the response instead of one in order to keep everything on even addresses. So in the working storage section, we have constructed a message and dedicated two bytes of memory:

```
TEMPORARY KEYBOARD INPUT
        RDATAB  2,0                    TEMPORARY KYBRD STORAGE
MENU OR ADD PROMPT
        DATA    13,13
        TEXT    'Do you wish to <A>dd another name or
                return to menu?'
```

In the above routine, please note that the text message appears on one complete line in the source code. (Due to limitations in the number of letters on a horizontal line in this book, we have shown it on two separate lines.)

The instruction set needed to display the message and get the input from the keyboard follows.

```
*ASK IF USER WANTS TO ADD ANOTHER OR RETURN TO MENU
ADD OR RETURN MENU OPTION
        MOVW    @A0,#12    *FUNCTION CODE
        MOVW    6@A0,#56   *# OF CHAR. IN PROMPT MESSAGE
        MOVW    8@A0,#1    *MAX # OF INPUT CHAR. ALLOWED
        MOVL    10@A0,#TEMPORARY KEYBOARD INPUT
        MOVL    14@A0,#MENU OR ADD PROMPT
        BRK     #0         *EXECUTE SUPERVISOR ROUTINE
```

We are on the last leg of our journey through the add routine. The only thing left is to test the character entered by the operator and take appropriate action.

The letter typed by the user is currently being held in the "temporary keyboard input" location. By loading that character into data register D0, he can compare them to check for the two appropriate responses. If an "A" has been selected, he wants to add another name. The program flow can then be directed to the "add menu" section where another name can be taken on. Should he want to return to the menu, the file must be closed and execution can branch to the top of the program. Another short routine is needed for this option. We will call it the "return to menu" section. In it, a call to the close subroutine followed by a branch to the start of the program is all that is needed.

The response testing section should be terminated with an unconditional branch back to the routine we just created to print the

message and get an input. This will "idiot proof" the routine against the user hitting a key other than the two we are looking for.

```
*TEST FOR AN "A" OR "R" RESPONSE FROM OPERATOR
        LDB        .D0,/TEMPORARY KEYBOARD INPUT
        CMPB       .D0,#65      *TEST FOR AN "A" INPUT
        BE         ADD MENU     *GO TO ADD ROUTINE
        CMPB       .D0,#82      *TEST FOR AN "R" INPUT
        BE         RETURN TO MENU
        BR         ADD OR RETURN MENU OPTION
RETURN  TO MENU
        CALL       CLOSE
        BR         START
```

THE PRINT LABELS SECTION

Let's take stock of what has been accomplished so far. We are getting cramps in our writing hand and you are probably getting bleary-eyed from looking at all these listings. But the program is beginning to take shape—it really does something now. The next logical section to tackle is the printout label section because now that we can put names into the machine, it would be nice to see all of the information stored in the file by printing it out on paper.

The "print labels" module of our program will be a little bit long. For that reason it is wise to first get some idea of how to attack this problem.

Each record in the file must be brought into the "disk buffer area", starting with the first record in the file and continuing through to the last record. So we will need to know the number of the first record that contains a name and the last. As you recall, we are using up the first record for name storage and just letting it keep track of the last used record in the file. Therefore, the first name on the list is stored in record number one. Record number zero is the first record in the file but not the first with a name in it. By reading record zero and getting the number stored there, we will know the number of the record that is at the end of the file.

Once a record is brought into the RAM area, the various parameters (name, address, city, and so forth) can be individually sent to the printer to form a label. Then a few line feeds can be sent to the printer to position the next label for printing. Since line feeds will have to be sent following each of the parameters and several line feeds at the end of each label, it would be foolish not to employ a "print line feed" subroutine.

So we need to keep track of a few things. We must know the beginning and end of the file, the file identification number, and the printing of each label.

When labels for all of the people on the list have been completed, the file has to be closed and the program flow should be directed back to the menu.

Let's start the routine and see what trouble we can get into by boldly pushing onward.

Using the same formula for printing a heading on the top lines of the video display, the text can be established in the "message and working storage area." The ASCII code of five is used to indicate the end of the message, just as it was in the add routine. This is a prerequisite for the routine, just as STORE1 must be set up as an entry condition. STORE1 can be loaded with the address of the printout screen text and we can call the screen display routine to get the message onto the video. Prior to that the clear screen routine can be utilized. Following the screen formatting, the name list file can be opened, again by the magic of subroutines.

```
**********************************
* MESSAGE & WORKING STORAGE AREA *
**********************************
PRINTOUT LABELS SCREEN DISPLAY
        TEXT     'PRINTOUT ADDRESS LABELS'
        DATAB    13
        TEXT     '-------------------------------------.
        TEXT     '-------------------------------------.
        DATAB    5,0
```

As always, to prevent the aggravating odd address trap error, we defined an extra byte after the ASCII five terminator. A zero is simply placed in that byte.

The beginning point of the printout routine must be labeled PRINTOUT ADDRESS LABELS because that is what we told the computer to jump to in the menu.

```
PRINTOUT ADDRESS LABELS
        CALL     CLEAR SCREEN
        LDA      .A3,PRINTOUT LABELS SCREEN DISPLAY
        STL      .A3/STORE1
        CALL     SCREEN DISPLAY ROUTINE
        CALL     OPEN EXISTING FILE
```

The number pointing to the last record in the file is now in the NEXT RECORD NUMBER storage area. We need a chunk of memory to act as a counter, keeping track of the record number to print. We know the first name is located in record number one. A STORE1 already exists and will be put to use in this routine. In the message and working storage area we can define another storage area, STORE2, and set aside four bytes, just like STORE1. Record

numbers can be up to FFFFFFFF hexadecimal, so a long word of storage is needed.

```
*END OF FILE RECORD # NOW LOCATED IN "NEXT RECORD NUMBER"
        LDL     .D0,#1    *LOAD D0 WITH 1ST RECORD #
        STL     .D0,/STORE2 *STORE FILE COUNTER
```

A giant loop is needed to test for the end of the file, get a record, print it, cue up the next label, and branch back to the end of file test. In this way STORE2 can be incremented after each record is printed. It can then be compared to the value in "next record number" to see if the last record has been reached.

Now we will start the loop, stopping to make comments as we go along. The beginning point of our loop is EOF TEST, that is, "end of file" check.

```
EOF TEST
        LDL     .A1,/NEXT RECORD NUMBER
        ADDL    .A1,#1    *ALLOW PRINTING OF LAST RECORD
        LDL     .D0,/STORE2
        CMPL    .D0,.A1   *REACHED EOF YET?
        BE      PRINTER FINISHED
```

What makes a person a computer programmer? Anyone can learn the various reserved words, syntax, and mnemonics of a language. But it takes some logical thought to make a program work. Now here we have to scratch our heads and think a little. Since we happened to place the end of file test at the beginning of the loop, we have to increment the value obtained from the "next record number" prior to testing. The value in the "next record number" location points to the last record in the file that contains a name. We want that name to be printed on a label. If we simply tested for this value being the last record number and then jumped out of the loop, this final record would never actually get printed. Follow that logic?

In a moment we will have to increment the value in STORE2 so that it is always pointing to the proper record number.

As you can see, A1 was loaded with the value of the last record number in the file and then increased by one to let the final record fall through and get printed. Data register D0 is loaded with the current record number count. If the two are equal, then all the names on the list have been printed on labels. Execution then branches to a section labeled PRINTER FINISHED where we can close the file and return to the menu—more on that in a minute after we complete this loop.

If this is not the last record, execution falls through to the next section which follows.

```
*READ A RECORD FROM THE DISK
        LDW     .A1,/FILE IDENTIFICATION NUMBER   *GET ID
        LDA     .A0,SVC BLOCK
        MOVW    @A0,#35     *IDENTIFYING FUNCTION CODE
        STW     .A1,6@A0    *PUT ID IN SVC BLOCK
        LDA     .A1,DISK BUFFER AREA1
        STL     .A1,8@A0    *PUT BUFFER ADDR. INTO SVC
        LDL     .D0,/STORE2  *GET RECORD COUNTER
        STL     .D0,12@A0   *PUT RECORD TO GET INTO SVC
        ADDW    .D0,#1      *INCREMENT RECORD #
        STL     .D0,/STORE2  *STORE FILE COUNTER
        MOVW    16@A0,#0     *UNLOCK RECORD
        BRK     #0
```

We just read a record from the disk and stuck it into the disk buffer area1 of our program. This is a lot simpler than creating another storage area. Each line there is already divided up with other labels for convenient distributing on the label.

This time, when we brought the value in STORE2 into a register for use with the read supervisor call, we incremented and stored the new value back again.

At this point it might be a good idea to set up a "flag" routine so we can see what is going on down there inside our machine (we think that there are really little men playin "TRON" games on our circuit boards). It is a simple matter to dump the entire disk buffer area1 onto the screen with one supervisor call. The entire 90-byte record can be thrown onto the video. It will look a little sloppy so you may not want to leave it in your final version but simply put it in as a debugging tool. Any carriage returns hit during entry of the name and address will also appear on the screen. This is good since it will help separate the name, address, town, and so on. As each record is obtained from the disk, it will appear on the screen.

```
*FLAG TO DUMP TO RECORD TO SCREEN
        LDA     .A0,SVC BLOCK
        MOVW    @A0,#9      *ID CODE
        MOVW    6@A0,#90    *LENGTH OF LINE
        MOVW    8@A0,#32    *SEND SPACE AT END OF ROUTINE
        MOVL    10@A0,#INPUT1   *ADDRESS
        BRK     #0
```

Again the time comes to put on an extra thinking cap. We have to develop a way to handle carriage returns that are present in the name, address, town, zip, and code lines. If the operator had hit the <ENTER> key after every item when he was entering the data, there would be no problem. We could simply print each line, or for that matter, we could just send the entire 90-byte buffer to the printer and the label would be printed.

However, as you will recall, we only allowed two spaces for

entering the state. Since all states have two-letter abbreviations, no carriage return can ever be stored here. If we allowed three characters for the state instead, the overall problem of handling line feeds/carriage returns would still not be solved. This is because it is still possible to make an entry for one of the other parameters, such as name, address, and so on and not hit <ENTER>. This would happen if the operator typed in an address, for example, which is 25 characters long. The input routines we used to get information from the keyboard are terminated by either the operator hitting <ENTER> or by his exceeding the maximum number of characters allowed for that particular item.

So we need to set up a loop that will test each character for a carriage return. This loop will resemble the display a character loop we have already written. In that routine, an ASCII code of five is tested and displaying of each character is done until the five is encountered. In this printer routine, we will test for a carriage return. Since some will and some will not have a carriage return, we want to be in control of when that code is sent to the printer. We should set up a subroutine that will print a line feed-carriage return. Then we can call it up as we need it. Another factor that must be kept track of here is the maximum allowable number of characters.

So there will be two things that we must test for in our loop. First we must test for a carriage return. If we find one, we know there are no more letters to be printed on that line. At that point, our own carriage return can be sent to the printer and the next line prepared for printing.

The other thing we must test for is the number of characters displayed. We do not want more characters displayed than the maximum number that was allowed during input.

Things get a little complicated in this routine since we are going to have two routines called from another. Perhaps it is advisable to sketch out a short flowchart of this section.

Figure 10-1 shows the part of the routine that will actually print a label. It does not include the section that reads a record from the disk and tests to see if it is the last record from the disk and tests to see if it is the last record in the file. It assumes that a record has already been read and is currently residing in the disk buffer area1 of the program. This will help clarify our thinking in writing this section of the program.

Follow the bouncing ball through the flowchart as we narrate. A counter is needed to keep tabs on the number of characters that are in the field; 25 for the name, 25 for address, and so on. Another
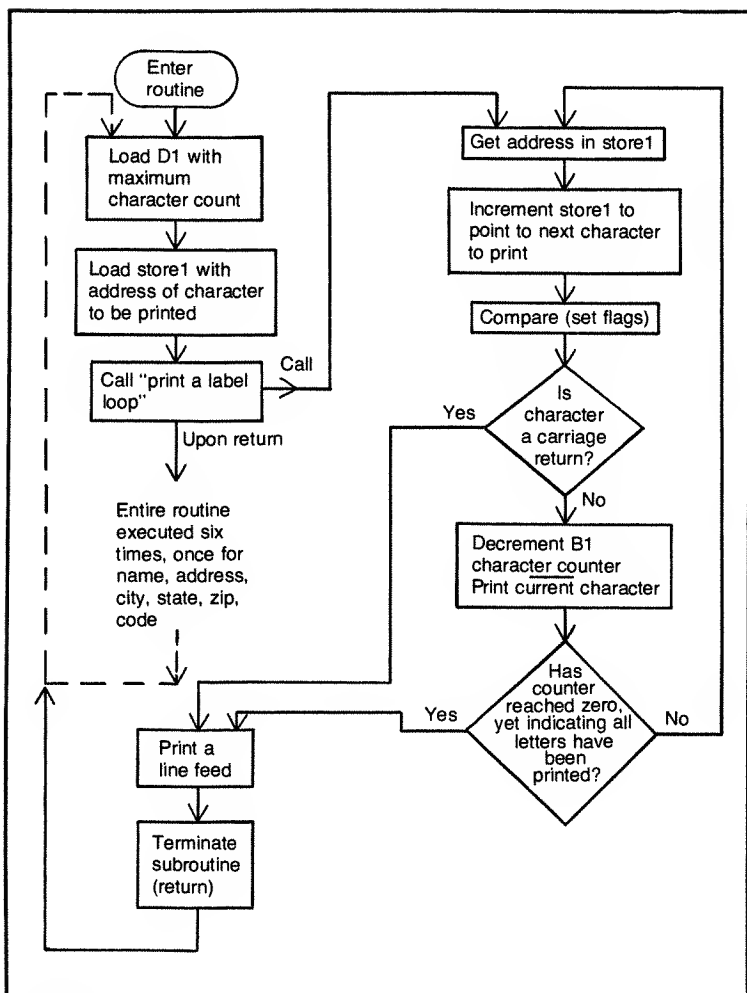
Fig. 10-1. Flowchart of print A label section.

storage area could be set up, but it may be best to avoid cluttering up the program with any more labels unless absolutely necessary. The MC68000 has over a dozen registers available, so we can temporarily put one of them to work.

We have been playing with data register D0 in the "read a record from the disk" section, so it is sensible to avoid it. Let's assign D1 the job of keeping track of the number of characters to print for each field. When the name line is to be printed, D1 can be loaded with a 25. A routine to dump a character to the printer can be

called. After that, the value in D1 can be decremented. If it should reach zero, then all 25 characters have been printed.

Our old buddy, STORE1 is up for grabs. At this point in the program, he has not been used yet. STORE1, since it is four bytes wide, can keep the address of the current character to be printed.

So, we have placed the maximum allowable character count into D1 and the address of the first character to be printed into STORE1. Now a routine to print the line (a field) can be called.

Once inside this call routine (given the name PRINT A LABEL LOOP in the program), the address saved in STORE1 can be placed into the SVC block in preparation for executing the supervisor call to send the character to the printer.

The value in STORE1 can then be incremented such that it will point to the address of the next character to be printed. Again, since some address registers are being used, let's pick, say, register A5 to make these moves for us.

We have decided to place all of the printer routines together in the same section of the program. If you wish, you could put the "print a label loop" into the subroutine area of the program. However, the program is beginning to get a little long. We thought it was best to keep all of the instructions involved with the printer together, since none of these subroutines will be needed anywhere else in the program.

So far, the subroutine that will print one line of the label looks like this:

```
PRINT A LABEL LOOP
        LDL       .A5,/STORE1
        MOVB      7@A0,@A5
        MOVB      6@A0,#0    *INSURES A ZERO THERE
        ADDW      .A5,#1    *INCREMENT
        STL       .A5,/STORE1   *STORE AWAY NEXT CHAR.
```

Before we send that character to the printer, we need to see if it is a carriage return or not. As mentioned previously, some fields may not contain a carriage return, such as the state. Therefore, we thought it best if we do not let any line feeds be printed that are imbedded within the disk buffer text. We want to be in control of all line feeds. When a line is done printing, we will send a carriage return to the printer. Otherwise, there would be two line feeds between those lines where the operator hit <ENTER> to end a field and one line feed when the <ENTER> key was not depressed.

In the listing where we just left off, 7@A0 is currently keeping the ASCII code of the character ready to go to the printer.

100

Let's move it into an unused data register, say D5, and compare it to an ASCII 13. If the character is a 13 then we do not print it. This means that all of the characters for that field have been entered by the operator. So we can print a line feed and bail out of this subroutine. That would mean going back to the place where D1 could be loaded with the next maximum character count and STORE1 set up to point to the location of the 1st character in the line to be printed.

Getting back to our comparison test, if the character to be printed is not a carriage return, then we can print that character. After that, the value in our character counter, D1, can be decremented. D1 can then be checked to see if all of the characters allowed in that field have been printed. If they have, then execution can branch to the routine we just described which prints a line feed and jumps back to get new values in D1 and STORE1 for another line of printing. If it fails the test, then the program can loop back to the place where the address in STORE1 is obtained and another test for a carriage return and printing takes place.

Continuing with the print a label loop, we add on these new instructions:

```
LDB     .D5,7@A0
CMPB    .D5,#13    *TEST FOR CARRIAGE RETURN
BE      PRINT LINE FEED
BRK     #0
CMPW    .D1,#0
SUBW    .D1,#1    *DECREMENT COUNTER
BE      PRINT LINE FEED
BR      PRINT A LABEL LOOP
```

The subroutine to print a line feed is a simple supervisor call to send an ASCII 13 to the printer.

```
*SUBROUTINE TO PRINT LINE FEED CARRIAGE RETURN
PRINT LINE FEED
        LDA     .A0,SVC BLOCK
        MOVW    @A0,#18    *IDENTIFYING CODE-PRINT CHAR.
        MOVW    6@A0,#13   *DUMP A LINE FEED TO PRINTER
        BRK     #0
        RET
```

In this program we will simply use brute force and manually set up each D1 and STORE1 after which a call to the "print a label loop" section will print the line. This could all be done in fewer steps, but it would make the program harder to follow. The thought here is to clarify the programming concepts. Once you learn the ropes, you can easily make improvements to this program.

When the label has been completely printed, several line feeds

are needed to cue up the printer to the next label. Labels are supplied on continuous rolls or on fan-folded style sheets.

Fortunately, a subroutine to print a line feed already exists. By calling that routine several times, the label is advanced by a couple of lines.

So the lines prior to the print a label loop, which call that routine six times (once for name, once for address, and so on through the code), will appear as follows:

```
*PRINT A LABEL
          LDA      .A0,SVC BLOCK
          MOVW     @A0,#18   *IDENTIFYING FUNCTION CODE
          LDA      .A5,INPUT1
          STL      .A5,/STORE1
          LDW      .D1,#25   *MAX # OF CHAR IN NAME LINE
          CALL     PRINT A LABEL LOOP
          LDA      .A5,INPUT2
          STL      .A5,/STORE1
          LDW      .D1,#25   *MAX # OF CHAR IN ADDR LINE
          CALL     PRINT A LABEL LOOP
          LDA      .A5,INPUT3
          STL      .A5,/STORE1
          LDW      .D1,#25   *MAX # OF CHAR IN TOWN LINE
          CALL     PRINT A LABEL LOOP
          LDA      .A5,INPUT4
          STL      .A5,/STORE1
          LDW      .D1,#2    *MAX # OF CHAR IN TOWN LINE
          CALL     PRINT A LABEL LOOP
          LDA      .A5,INPUT5
          STL      .A5,/STORE1
          LDW      .D1,#8    *MAX # OF CHAR IN ZIP LINE
          CALL     PRINT A LABEL LOOP
          LDA      .A5,INPUT6
          STL      .A5,/STORE1
          LDW      .D1,#5    *MAX # OF CHAR IN CODE LINE
          CALL     PRINT A LABEL LOOP
          CALL     PRINT LINE FEED   *CUE UP FOR NEXT LABEL
          CALL     PRINT LINE FEED
          BR       EOF TEST
```

At the beginning of the printer routine, we discussed the section to test for the end of file. When all of the labels have been printed, we instructed the machine to go to the address labeled PRINTER FINISHED. Now it is time to create that section.

The only thing which remains is to close the file and jump to the main menu. Since we only have one file to worry about, we can use the blanket close supervisor call, number 133. Then execution can branch back to the start of the whole program, where the menu will again be displayed.

```
PRINTER FINISHED
          LDA      .A0,SVC BLOCK
          MOVW     @A0,#133  *ID TO CLOSE ALL FILES
          BRK      #0
          BR       START
```

102

That completes the print labels section of the program.

THE SEARCH AND MODIFY ROUTINES

While it is feasible to write search and modify routines independently of each other, many lines of programming can be eliminated if we integrate these two functions.

Before it is possible to modify a person on file, you first must search for him to see if he is on the disk. If he is, then the whole record must be read into memory where we can examine it.

The search routine will simply require us to ask the operator the name of the person to look up. At that point the computer must scan the entire list. When the name is found, the entire record will need to be shown on the video display. When the operator is done studying it, he can indicate that he is done and wants to return to the main menu.

First, both the search and modify sections should follow the traditional pattern that we have established. Upon entering these routines, the screen should be cleared and the appropriate heading should be shown in order to confirm for the user that he is indeed in the function that he actually wants.

The search section can start with the following:

```
SEARCH ROUTINE
*SETUP SEARCH SCREEN DISPLAY
        CALL    CLEAR SCREEN
        LDA     .A3,SEARCH SCREEN DISPLAY
        STL     .A3,/STORE1
        CALL    SCREEN DISPLAY ROUTINE
```

As you can see, we set up the entry conditions prior to calling the screen display routine. The address of the first text character to print is loaded into STORE1. So down in the message and working storage area we define the search page heading.

```
SEARCH SCREEN DISPLAY
        TEXT    'SEARCH FOR A NAME'
        DATAB   13
        TEXT    '-------------------------------------------'
        TEXT    '-------------------------------------------'
        DATAB   13,13
        DATAB   5,0    *TERMINATOR CHARACTER
```

The modify section can start with the same idea, but since both the search and modify routines need to search through the file for the name, why not just use one routine?

That is exactly what can be done. The modify section can branch over to the search routine once the heading has been set up.

However, the search routine should know whether execution was sent there by the operator selecting the search option or by the modify option. Therefore, we will let a register be a "flag," that is, an indicator to show that the modify routine is sending the execution to the search section. By checking this flag, we will know where to branch to after the name has been found. If the search section sent it there, then execution should merely return to the menu. Otherwise, the operator will have to re-enter the information in that record, updating or changing it.

Again we could pick a storage area, but with so many registers to work with, let's choose D6. Arbitrarily we have decided to place a value of one in D6 if the modify option was selected at the menu, and a zero there if search did.

After that, both routines need to get a name from the keyboard for which it can begin searching. How about giving it the label ASK OPERATOR NAME TO SEARCH FOR?

```
MODIFY ROUTINE
*SETUP MODIFY SCREEN DISPLAY
          CALL      CLEAR SCREEN
          LDA       .A3,MODIFY SCREEN DISPLAY
          STL       .A3,/STORE1
          CALL      SCREEN DISPLAY ROUTINE
          LDL       .D6,#1    *MODIFY "FLAG"
          BR        ASK OPERATOR NAME TO SEARCH FOR

(add to the MESSAGE AND WORKING STORAGE AREA:)

MODIFY SCREEN DISPLAY
          TEXT      'MODIFY A RECORD'
          DATAB     13
          TEXT      '-------------------------------------------'
          TEXT      '-------------------------------------------'
          DATAB     13,13
          DATAB     5,0    *TERMINATOR CHARACTER
```

What happens if the operator selects the modify option, re-types the name correctly, is returned back to the menu, and then selects the search option? At the moment, once D6 is set to a value of one, it is never reset to zero. So anytime an operator chooses to search for someone after he has first used the modify routine, he will again be forced into the modify section.

To prevent this from happening, one of the first lines of the program should reset the flag to zero. In this way, everytime the menu is shown, the modify flag will be reset. So between the START and CALL CLEAR SCREEN instructions, insert

```
          LDL       .D6,#0    *D6 USED AS MODIFY FLAG-RESET
```

104

The next two sets of instructions will prompt the operator to enter the name of the person he wishes to search for. A keyboard input routine can get the name and place it into a buffer area.

Once the name the operator wants is in memory, we can pull records from the disk one at a time and see if the two names match.

The CoMPare instructions has a neat variation where it can examine two characters stored in memory and perform a "post increment" function. In a moment, we will show how this powerful 16-bit mnemonic can save a lot of work.

In order to create a loop around the routine that compares each byte, we will need to know how many characters the operator has typed in. In that way, if the name on the disk is "Smith" and the operator only types in "Smi" then a match will be found. This is nice in that the operator only has to enter as many letters as necessary to distinguish that name from any other on the list. It also insures that a match will be made if the name is on the list. You see, we have allowed 25 bytes of memory for the name. The disk stores 25 bytes whether the name is 25 letters long or not. Therefore, it would be conceivable that a match may never be made if the operator entered "Smith", which is 5 letters long, and the disk buffer contained the letters "Smith" plus 20 more spaces. As you will recall, prior to adding a name in the add routine, we fill the buffer with spaces (ASCII 32). So, only the number of characters entered will be checked for a match.

Now we need to display a message and get a response from the operator. This time we cannot use the supervisor call that performs both of these functions (with a function code of nine) since it does not return to us the length of the string of characters entered.

The display a line supervisor call and the keyboard line call (function code five) will have to be used.

We can label the search message SEARCH PROMPT and the keyboard input can go under SEARCH NAME INPUT, which we will have to define in the working storage area.

Since STORE1 stands ready, willing and able to serve us, we can get the number of characters entered via the keyboard (stored in   12 @ A0   upon exit) and stick it in STORE1 for safe keeping. After than, we'll figure out what to do with it.

```
ASK OPERATOR NAME TO SEARCH FOR
        LDA     .A0,SVC BLOCK
        MOVW    @A0,#9    *IDENTIFYING FUNCTION CODE
        MOVW    6@A0,#26  *# OF CHAR IN MESSAGE
        MOVW    8@A0,#32  *SEND A SPACE AFTER TEXT
        MOVL    10@A0,#SEARCH PROMPT
```

```
        BRK       #0
*GET KEYBOARD INPUT
        MOVW      @A0,#5    *IDENTIFYING FUNCTION CODE
        MOVW      6@A0,#25  *# OF ALLOWABLE INPUT
        MOVL      8@A0,#SEARCH NAME INPUT
        BRK       #0
        LDW       .A1,12@A0   *GET # OF CHAR. ENTERED
        STW       .A1,/STORE1   *PUT IN STORE1

(add to message and working storage area:)

SEARCH PROMPT
        TEXT      'Enter name to search for: '
SEARCH NAME INPUT
        RDATAB   26,0
```

Now we can open the file. We will jump to the open existing file routine since no one would ever want to search for a name before any names have been added. However, you may wish to place an error trap here in your own version of this program in order to catch an operator who may try to select the search or modify option without first using the add routine to put at least one name on file.

The last record number is placed into "next record number" by the open routine. We will assign STORE2 the task of keeping count of the record number we have read.

The value in STORE2 can then be compared to the value in the "next record number" to detect when the entire list has been scanned. If the two are equal, then the whole file has been examined. We then branch to "search finished." These instructions close the file and jump the program flow back to "start". You may want to add a message here for the operator, perhaps something to the effect "Name not found."

This whole section basically has the same structure as the one we developed for the print label routine.

If we are not at the end of the file, a record can be read from the disk. So far we have

```
*OPEN FILE (MUST ALREADY EXIST)
        CALL      OPEN EXISTING FILE
*END OF FILE RECORD # NOW LOCATED IN "NEXT RECORD NUMBER"
        LDL       .D0,#1    #LOAD D0 WITH 1ST RECORD
        STL       .D0,/STORE2     *STORE FILE COUNTER
END OF FILE TEST
        LDL       .A1,/NEXT RECORD NUMBER
        ADDL      .A1,#1
        LDL       .D0,/STORE2
        CMPL      .D0,.A1    *REACH END OF FILE YET?
        BE        SEARCH FINISHED
*READ A RECORD FROM THE DISK
        LDW       .A1,/FILE IDENTIFICATION NUMBER
        LDA       .A0,SVC BLOCK
        MOVW      @A0,#35    *IDENTIFYING FUNCTION CODE
        STW       .A1,6@A0   *PUT ID IN SVC BLOCK
```

```
LDA      .A1,DISK BUFFER AREA1
STL      .A1,8@A0  *PUT BUFFER ADDR INTO SVC
LDL      .D0,/STORE2   *GET RECORD COUNTER
STL      .D0,12@A0     *PUT RECORD # TO GET INTO SVC
ADDW     .D0,#1    *INCREMENT RECORD #
STL      .D0,/STORE2   *STORE FILE COUNTER
MOVW     16@A0,#0    *UNLOCK RECORD
BRK      #0
```

Now for that neat new variation of CoMPare. Currently we have two blocks of memory which we want to test byte by byte for equality. By loading two address registers with the memory location of the beginning character in each block, we can take advantage of the post increment ability of the CMP op code.

CMP as we shall use it, will compare a byte in one memory location with that stored in another. The appropriate flags in the status register will be set or cleared, depending upon the results of the comparison. By using the plus sign (+), the computer automatically increments the value stored in each register by one. In this way the addresses currently in the two registers are increased by one byte such that they now point to the next character in each block. This command is another example of how the MC68000 instruction set is capable of doing several operations in one step as compared to eight-bit microprocessors.

Now we load A3 with the address of the name and the user wants to look up. As each record is pulled in and placed into "disk buffer area "1", the name from the disk file will be located at "input 1." A loop can check each character in the two names to see if they all match.

Several things need to be kept account of in this loop. First, our character counter, D1, is decremented every time a pass of the loop is made. When it reaches zero, the number of characters that the operator typed in has been checked. If it does reach zero, then a total match has been made and execution can jump out of the loop into the "found match" routine (which we have yet to invent).

Should any character not match, then the current record is not the one we are looking for. Another read from the disk can be done and the next record brought in and examined.

```
*CHECK NAME FROM RECORD AND FROM DISK FOR MATCH
        LDW      .D1,/STORE1     *D1 IS CHAR. COUNTER
        LDA      .A2,SEARCH NAME INPUT    *GET ENTERED NAME
        LDA      .A3,INPUT1      *GET NAME ON DISK
SEARCH LOOP
        SUBW     .D1,#1    *DECREMENT CHAR. COUNTER
        BE       FOUND MATCH    *ALL CHARACTERS MATCHED
        CMPB     @A2+,@A3+    *COMPARE A LETTER
BE      SEARCH LOOP    *MATCH-CHECK NEXT LETTER
BR      END OF FILE TEST    *GET ANOTHER RECORD
```

Once we have found the name, we can display all of the fields on the screen. For the sake of simplicity we will use the display a line supervisor call to get the full contents of the record onto the video tube. However, you may want to dress up this section of the program. Remember, some fields contain line feeds in them and some do not, so this makes the printout a little ragged looking. In any event, the routine to display the name is next.

```
FOUND MATCH
*DISPLAY NAME
        LDA       .A0,SVC BLOCK
        MOVW      @A0,#9      *DISPLAY A LINE ID CODE
        MOVW      6@A0,#25    *25 CHARACTERS IN THE NAME
        MOVW      8@A0,#13    *SEND LINE FEED AT END
        MOVL      .10@A0,#INPUT1
        BRK       #0
*DISPLAY ADDRESS
        MOVW      6@A0,#25    *25 CHARACTERS IN THE ADDR.
        MOVL      10@A0,#INPUT2
        BRK       #0
*DISPLAY CITY/TOWN
        MOVW      6@A0,#25    *25 CHARACTERS IN THE TOWN
        MOVL      10@A0,#INPUT3
        BRK       #0
*DISPLAY STATE
        MOVW      6@A0,#2     *2 CHARACTERS IN THE STATE
        MOVL      10@A0,#INPUT4
        BRK       #0
*DISPLAY ZIP
        MOVW      6@A0,#8     *8 CHARACTERS IN THE ZIP
        MOVL      10@A0,#INPUT5
        BRK       #0
*DISPLAY CODE
        MOVW      6@A0,#5     *5 CHARACTERS IN THE CODE
        MOVL      10@A0,#INPUT6
        BRK       #0
```

We must check here to see if the modify option was selected. If it was, we can jump to a new section, "modify selected". If not, execution will fall through to the next part where the operator can hit the <ENTER> key when he is done examining the data on the screen. Then he can be returned to the menu. We will set it up such that it only allows a one letter response so the user can hit <ENTER> or actually any key he wants to terminate the routine. This is a good place to stick the SEARCH FINISHED instructions that we mentioned a moment ago.

```
*CHECK TO SEE IF MODIFY OPTION WAS SELECTED
        CMPB       .D6,#1    *1 INDICATES MODIFY
        BE         MODIFY SELECTED
*IF FALLS THROUGH, THEN END OF SEARCH ROUTINE
        MOVW       @A0,#12    *ID CODE VIDKEY
        MOVW       6@A0,#38   *# OF CHAR IN PROMPT MESSAGE
        MOVW       8@A0,#1    *LENGTH OF ALLOWABLE INPUT
        MOVL       10@A0,#TEMPORARY KEYBOARD INPUT
```

```
        MOVL        14@A0,#HIT ENTER PROMPT
        BRK         #0
        BR          SEARCH FINISHED
SEARCH FINISHED
        MOVW        @A0,#133   *ID TO CLOSE ALL FILES
        BRK         #0
        BR          START
```

(add to the MESSAGE AND WORKING STORAGE AREA:)

```
HIT ENTER PROMPT
        TEXT        'Hit the <ENTER> key to return to menu
```

It actually is not necessary for us to use the BR SEARCH FINISHED instruction since we follow it with that routine. But we put it in there in the event that you may make some changes to the program and slip more instructions into that area.

We are down to the last few lines of our program (aren't you glad!). The "modify selected" portion can actually make use of the "add a name" section of our program.

But first we must get the current record number stored in "next record number". If you follow the logic of how the record number was placed there, you will see that the value currently in that storage area is actually pointing two records too high. This is easily remedied by subtracting two from it. By saving this value in "next record number", we can jump to the "add menu" where the operator can retype the entire name, address, city, and so on. This is the easiest thing to do. The record to write this to is ready and waiting in the "next record number" area, so the write routine will automatically put it just where we want it.

```
MODIFY SELECTED
*GET THE CURRENT RECORD #,STORE IN 'NEXT RECORD NUMBER'
        LDL         .D0,/STORE2
*AT THIS POINT, TWO RECORDS HIGHER SO MUST DECREMENT
        SUBL        .D0,/NEXT RECORD NUMBER
        BR          ADD MENU
```

Our add routine allows you to add as many names as you wish without ever having to go back to the main menu. While it is fine for our modify function to use the add routine, we do not want the prompt "Do you wish to <A>dd another name or <R>eturn to menu?" to be seen. It would be a good idea if we let the operator modify several names without having to go back to the menu just as we did in the add routine. This can be done easily with the addition of only a few lines of programming.

After the write section in the add routine, test to see if the modify routine sent it there. If it did, D6, our "modify flag" will contain a one. Should D6 equal one, branch to the search routine

where we can again search for another name.

```
*CHECK TO SEE IF MODIFY ROUTINE SENT IT HERE
        CMPB    .D6,#1   *IS FLAG SET?
        BE      SEARCH ROUTINE
```

This is fine; however, the search routine opens a file, and at this point, the file is already open! Again, we need to add a line for the fix. Locate the search routine and insert the following to make it the first line in that section.

CALL   CLOSE   *MODIFY ROUTINE NEEDS THIS

Now the operator is locked into an endless loop of modifying names, searching for another, modifying that, etc. We need a way to allow the user to bail out of the modify routine and return to the menu.

By allowing a bailout right at the input of the name to search for, this will also allow the user to exit the search routine without actually having to search for anyone.

Instruct the operator to simply hit the <ENTER> key in response to "Enter name to search for:." Since there can never be just one character entered for a name, all we have to do is check the number of keys hit and see if it is a one. Even if a person's name is only one letter (highly unlikely), it would still require two keys to be depressed, one for the letter and one for the <ENTER> key.

Locate the "*get keyboard input" section of our add routine. You will see that after the supervisor call is executed, the number of entered characters is returned into byte-offset 14 of the SVC block. Note that we temporarily used register A1 to pull the value in 14@A0 and place it into STORE1. Therefore, A1 still holds that number. After the STW .A1,/STORE1 instruction, we insert the following few lines to complete the program.

```
*TEST TO SEE IF GUY JUST HIT <ENTER> INSTEAD OF NAME
*TO ALLOW FOR BAIL OUT-CAN MODIFY MORE THAN ONE NAME
        CMPW    .A1,#1    *JUST ONE CHARACTER RETURNED
        BE      START     *BAIL OUT TO MENU
```

WRAPPING IT ALL UP

That's the basic structure for any type of simple disk storage program. It can be endlessly improved upon, as can any program. There always seems to be just one more feature that you can incorporate. The main idea however, was to keep this program as short and as clear as any machine-language program can be.

110

We did achieve the concept of modular programming, although the program could certainly be reduced even further in size by some thoughtful reworking of a few sections. More subroutines could be employed. Also, the whole program could be set up as a series of calls. For example, we used the long method of getting a name, address, city, and so on from the keyboard. One routine could be set up that would get all of these inputs from the operator by calling the same supervisor call. Before calling it, the various parameters could be put in place similar to what we did for the display a character routine.

The best programming rule is never to have anything in your program more than once—make a subroutine out of it. This could certainly be done with the constant text defining of hyphens for use with the video screen headings:

```
TEXT       '-----------------------------------------'
```

Another suggestion we may make would be to expand on the modify routine. Currently, the operator must retype the entire record. It would take a little extra programming effort, but it could be altered so that the operator would only need to retype the field or fields that he needs to change.

The printer routine is also a candidate for rewriting. Presently, each field is printed on its own line. In actual practice you may decide not to print a line feed between the city, state, and zip to put it more in keeping with the way we usually address envelopes.

No provision has been made for deleting a name, but this is an idea which could be implemented. In many business applications, once a person is on a list, they never want him erased. In other circumstances, an active file may need to constantly add and delete entries. If this is the case, some thought would have to be given to the recovery of file space, that is, the filling in of records where a person has been deleted.

MAILING LIST SOURCE CODE

The entire source code listing all pieced together follows. Line numbers are given to make entering the program into the computer more convenient.

```
 1  START
 2          LDL     .D6,#0          *D6 USED AS MODIFY FLAG-RESET
 3          CALL    CLEAR SCREEN
 4  POSITION CURSOR
 5          MOVW    @A0,#10         *POSITION CURSOR FUNCTION CODE
 6          MOVW    6@A0,#1         *ROW POSITION (TOP TO BOTTOM)
 7          MOVW    8@A0,#38        *COLUMN POSITION (LEFT TO RIGHT)
 8          BRK     #0
 9  INITIAL SETUP
10          LDA     .A3,MENU
11          STL     .A3,/STORE1     *STORES AWAY BEGINNING OF MENU TEXT ADDRESS
12          CALL    SCREEN DISPLAY ROUTINE
13  KEYBOARD INPUT ROUTINE
14          MOVW    @A0,#5          *FUNCTION CODE
15          MOVW    6@A0,#1         *NUMBER OF CHARACTERS TO BE ENTERED
16          MOVL    8@A0,#KEYBOARD STORE1
17          BRK     #0
18  *TEST INPUT
19          LDB     .D0,/KEYBOARD STORE1
20          CMPB    .D0,#65         *TEST FOR AN "A" INPUT
21          BE      ADD ROUTINE
22          CMPB    .D0,#83         *TEST FOR AN "S" INPUT
23          BE      SEARCH ROUTINE
24          CMPB    .D0,#77         *TEST FOR A "M" INPUT
25          BE      MODIFY ROUTINE
```

```
26          CMPB      .D0,#80    *TEST FOR A "P"
27          BE        PRINTOUT   ADDRESS LABELS
28          CMPB      .D0,#82    *TES FOR AN "R"
29          BE        JUMPDOS
30          BR        START
31 ADD ROUTINE
32          CALL      OPEN
33 ADD MENU
34          CALL      CLEAR SCREEN
35          LDA       .A3,ADD SCREEN DISPLAY
36          STL       .A3,/STORE1
37          CALL      SCREEN DISPLAY ROUTINE
38 *GET NAME FROM KEYBOARD
39          CALL      CLEAR BUFFER
40          MOVW      @A0,#12    *FUNCTION CODE-DSPLY TEXT & GET INPUT
41          MOVW      6@A0,#14   *# OF CHAR. IN PROMPT MESSAGE
42          MOVW      8@A0,#25   *MAX # OF INPUT CHAR. ALLOWED
43          MOVL      10@A0,#INPUT1
44          MOVL      14@A0,#PROMPT1
45          BRK       #0
46 *GET STREET ADDRESS FROM KEYBOARD
47          MOVW      6@A0,#24   *# OF CHAR. IN PROMPT MESSAGE
48          MOVW      8@A0,#25   *MAX # OF INPUT CHAR. ALLOWED
49          MOVL      10@A0,#INPUT2
50          MOVL      14@A0,#PROMPT2
```

```
51          BRK     #0
52  *GET TOWN/CITY FROM KEYBOARD
53          MOVW    6@A0,#22    *# OF CHAR. IN PROMPT MESSAGE
54          MOVW    8@A0,#25    *MAX # OF INPUT CHAR. ALLOWED
55          MOVL    10@A0,#INPUT3
56          MOVL    14@A0,#PROMPT3
57          BRK     #0
58  *GET STATE FROM KEYBOARD
59          MOVW    6@A0,#36    *# OF CHAR. IN PROMPT MESSAGE
60          MOVW    8@A0,#2     *MAX # OF INPUT CHAR. ALLOWED
61          MOVL    10@A0,#INPUT4
62          MOVL    14@A0,#PROMPT4
63          BRK     #0
64  *GET ZIP FROM KEYBOARD
65          MOVW    6@A0,#18    *# OF CHAR. IN PROMPT MESSAGE
66          MOVW    8@A0,#8     *MAX # OFINPUT CHAR. ALLOWED
67          MOVL    10@A0,#INPUT5
68          MOVL    14@A0,#PROMPT5
69          BRK     #0
70  *GET SPECIAL CODE FROM KEYBOARD
71          MOVW    6@A0,#32    *# OF CHAR. IN PROMPT MESSAGE
72          MOVW    8@A0,#5     *MAX # OF INPUT CHAR. ALLOWED
73          MOVL    10@A0,#INPUT6
74          MOVL    14@A0,#PROMPT6
75          BRK     #0
```

```
76  WRITE
77          MOVW    @A0,#44         *IDENTIFYING FUNCTION CODE
78          LDW     .A1,/FILE IDENTIFICATION NUMBER      *PUT FILE ID IN A1
79          STW     .A1,6@A0        *PUT FILE ID # INTO SVC BLOCK
80          LDA     .A2,DISK BUFFER AREA1
81          STL     .A2,8@A0        *PUT BUFFER ADDR INTO SVC BLOCK
82          LDL     .A3,/NEXT RECORD NUMBER
83          ADDW    .A3,#1          *INCREMENT NEXT RECORD
84          STL     .A3,12@A0       *PUT RECORD # TO WRITE INTO SVC
85          STL     .A3,/NEXT RECORD NUMBER      * SAVE NEXT RECORD # POIN
86          BRK     #0              *EXECUTE SUPERVISOR ROUTINE
87  *CHECK TO SEE IF MODIFY ROUTINE SENT IT HERE
88          CMPB    .D6,#1          *IS FLAG SET?
89          BE      SEARCH ROUTINE
90  *SAVE NEXT AVAILABLE RECORD # IN RECORD #0
91          LDA     .A2,NEXT RECORD NUMBER
92          STL     .A2,8@A0
93          MOVL    12@A0,#0        *RECORD # 0 TO WRITE
94          BRK     #0
95  *ASK IF USER WANTS TO ADD ANOTHER NAME OR RETURN TO MENU
96  ADD OR RETURN MENU OPTION
97          MOVW    @A0,#12         *FUNCTION CODE-DSPLY TEXT & GET INPUT
98          MOVW    6@A0,#56        *# OF CHAR. IN PROMPT RESPONSE
99          MOVW    8@A0,#1         *MAX # OF INPUT CHAR. ALLOWED
100         MOVL    10@A0,#TEMPORARY KEYBOARD INPUT
```

```
101          MOVL    14@A0,#MENU OR ADD PROMPT
102          BRK     #0            *EXECUTE SUPERVISOR ROUTINE
103 *TEST FOR AN "A" OR "R" RESPONSE FROM OPERATOR
104          LDB     .D0,/TEMPORARY KEYBOARD INPUT
105          CMPB    .D0,#65       *TEST FOR AN "A" INPUT
106          BE      ADD MENU      *GO TO ADD ROUTINE ON "A" RESPONSE
107          CMPB    .D0,#82       *TEST FOR AN "R" INPUT
108          BE      RETURN TO MENU
109          BR      ADD OR RETURN MENU OPTION
110 RETURN TO MENU
111          CALL    CLOSE
112          BR      START
113 SEARCH ROUTINE
114 *SETUP SEARCH SCREEN DISPLAY
115          CALL    CLOSE         *MODIFY ROUTINE NEEDS THIS
116          CALL    CLEAR SCREEN
117          LDA     .A3,SEARCH SCREEN DISPLAY
118          STL     .A3,/STORE1
119          CALL    SCREEN DISPLAY ROUTINE
120 ASK OPERATOR NAME TO SEARCH FOR
121          LDA     .A0,SVC BLOCK
122          MOVW    @A0,#9        *IDENTIFYING FUNCTION CODE
123          MOVW    6@A0,#26      *# OF CHAR IN MESSAGE
124          MOVW    8@A0,#32      *SEND A SPACE AFTER TEXT
125          MOVL    10@A0,#SEARCH PROMPT
```

```
126    BRK      #0
127 *GET KEYBOARD INPUT
128    MOVW     @A0,#5        *IDENTIFYING FUNCTION CODE
129    MOVW     6@A0,#25      *# OF ALLOWABLE INPUT
130    MOVL     8@A0,#SEARCH NAME INPUT
131    BRK      #0
132    LDW      .A1,12@A0     *GET # OF CHAR ENTERED
133    STW      .A1,/STORE1   *PUT IN STORE1
134 *TEST TO SEE IF GUY JUST HIT <ENTER> INSTEAD OF NAME
135 *TO ALLOW FOR BAIL OUT-CAN MODIFY MORE THAN ONE NAME
136    CMPW     .A1,#1        *JUST ONE CHARACTER RETURNED
137    BE       START         *BAIL OUT TO MENU
138 *OPEN FILE (MUST ALREADY EXIST)
139    CALL     OPEN EXISTING FILE
140 *END OF FILE RECORD # NOW LOCATED IN "NEXT RECORD NUMBER"
141    LDL      .D0,#1        *LOAD D0 WITH 1ST RECORD
142    STL      .D0,/STORE2   *STORE FILE COUNTER
143 END OF FILE TEST
144    LDL      .A1,/NEXT RECORD NUMBER
145    ADDL     .A1,#1
146    LDL      .D0,/STORE2
147    CMPL     .D0,.A1       *REACH END OF FILE YET?
148    BE       SEARCH FINISHED
149 *READ A RECORD FROM THE DISK
150    LDW      .A1,/FILE IDENTIFICATION NUMBER   *GET FILE ID
```

117

```
151           LDA     .A0,SVC BLOCK    *IDENTIFYING FUNCTION CODE
152           MOVW    @A0,#35          *PUT ID IN SVC BLOCK
153           STW     .A1,6@A0         *DISK BUFFER AREA1
154           LDA     .A1,DISK BUFFER AREA1
155           STL     .A1,8@A0         *PUT BUFFER ADDR.INTO SVC
156           LDL     .D0,/STORE2      *GET RECORD COUNTER
157           STL     .D0,12@A0        *PUT RECORD # TO GET INTO SVC
158           ADDW    .D0,#1           *INCREMENT RECORD #
159           STL     .D0,/STORE2      *STORE FILE COUNTER
160           MOVW    16@A0,#0         *UNLOCK RECORD
161           BRK     #0
162 *CHECK NAME FROM RECORD AND FROM DISK FOR MATCH
163           LDW     .D1,/STORE1      *D1 IS CHAR. COUNTER
164           LDA     .A2,SEARCH NAME INPUT *GET ENTERED NAME
165           LDA     .A3,INPUT1       *GET NAME ON DISK
166 SEARCH LOOP
167           SUBW    .D1,#1           *DECREMENT CHAR.COUNTER
168           BE      FOUND MATCH      *ALL CHARACTERS MATCHED
169           CMPB    @A2+,@A3+        *COMPARE A LETTER
170           BE      SEARCH LOOP      *MATCH-CHECK NEXT LETTER
171           BR      END OF FILE TEST *GET ANOTHER RECORD
172 FOUND MATCH
173 *DISPLAY NAME
174           LDA     .A0,SVC BLOCK
175           MOVW    @A0,#9           *DISPLAY A LINE ID CODE
```

```
176        MOVW   6@A0,#25    *25  CHARACTERS  IN  THE  LINE
177        MOVW   8@A0,#13    *SEND LINE FEED AT END
178        MOVL   10@A0,#INPUT1
179        BRK    #0
180  *DISPLAY ADDRESS
181        MOVW   6@A0,#25    *25  CHARACTERS  IN  THE  ADDR.
182        MOVL   10@A0,#INPUT2
183        BRK    #0
184  *DISPLAY CITY/TOWN
185        MOVW   6@A0,#25    *25  CHARACTERS  IN  THE  TOWN
186        MOVL   10@A0,#INPUT3
187        BRK    #0
188  *DISPLAY STATE
189        MOVW   6@A0,#2     *25  CHARACTERS  IN  THE  STATE
190        MOVL   10@A0,#INPUT4
191        BRK    #0
192  *DISPLAY ZIP
193        MOVW   6@A0,#8     *8  CHARACTERS  IN  THE  ZIP
194        MOVL   10@A0,#INPUT5
195        BRK    #0
196  *DISPLAY CODE
197        MOVW   6@A0,#5     *5  CHARACTERS  IN  THE  CODE
198        MOVL   10@A0,#INPUT6
199        BRK    #0
200  *CHECK TO SEE IF MODIFY OPTION WAS SELECTED
```

119

```
201      CMPB      .D6,#1      *1 INDICATES MODIFY
202      BE        MODIFY SELECTED
203  *IF FALLS THROUGH, THEN END OF SEARCH ROUTINE
204      MOVW      @A0,#12      *ID CODE VIDKEY
205      MOVW      6@A0,#38     *# OF CHAR. IN PROMPT MESSAGE
206      MOVW      8@A0,#1      *LENGTH OF ALLOWABLE INPUT
207      MOVL      10@A0,#TEMPORARY KEYBOARD INPUT
208      MOVL      14@A0,#HIT ENTER PROMPT
209      BRK       #0
210      BR        SEARCH FINISHED
211  SEARCH FINISHED
212      MOVW      @A0,#133     *ID TO CLOSE ALL FILE
213      BRK       #0
214      BR        START
215  MODIFY SELECTED
216  *GET THE CURRENT RECORD #,STORE IN 'NEXT RECORD NUMBER'
217      LDL       .D0,/STORE2
218  *AT THIS POINT,TWO RECORDS HIGHER SO MUST DECREMENT
219      SUBL      .D0,#2
220      STL       .D0,/NEXT RECORD NUMBER
221      BR        ADD MENU
222  MODIFY ROUTINE
223  *SETUP MODIFY SCREEN DISPLAY
224      CALL      CLEAR SCREEN
225      LDA       .A3,MODIFY SCREEN DISPLAY
```

```
226  STL   .A3,/STORE1
227  CALL  SCREEN DISPLAY ROUTINE
228  LDL   .D6,#1            *MODIFY FLAG
229  BR    ASK OPERATOR NAME TO SEARCH FOR
230 PRINTOUT ADDRESS LABELS
231  CALL  CLEAR SCREEN
232  LDA   .A3,PRINTOUT LABELS SCREEN DISPLAY
233  STL   .A3,/STORE1
234  CALL  SCREEN DISPLAY ROUTINE
235  CALL  OPEN EXISTING FILE
236 *END OF FILE RECORD # NOW LOCATED IN "NEXT RECORD NUMBER"
237  LDL   .D0,#1            *LOAD D0 WITH 1ST RECORD #
238  STL   .D0,/STORE2       *STORE FILE COUNTER
239 EOF TEST
240  LDL   .A1,/NEXT RECORD NUMBER
241  ADDL  .A1,#1            *ALLOW PRINTING OF LAST RECORD
242  LDL   .D0,/STORE2
243  CMPL  .D0,.A1           *REACH END OF FILE YET?
244  BE    PRINTER FINISHED
245 *READ A RECORD FROM THE DISK
246  LDW   .A1,/FILE IDENTIFICATION NUMBER   *GET FILE ID
247  LDA   .A0,SVC BLOCK
248  MOVW  @A0,#35           *IDENTIFYING FUNCTION CODE(READ)
249  STW   .A1,6@A0          *PUT ID IN SVC BLOCK
250  LDA   .A1,DISK BUFFER AREA1
```

```
251        STL    .A1,8@A0    *PUT BUFFER ADDR. INTO SVC
252        LDL    .D0,/STORE2   *GET RECORD COUNTER
253        STL    .D0,12@A0   *PUT RECORD # TO GET INTO SVC
254        ADDW   .D0,#1           *INCREMENT RECORD # *
255        STL    .D0,/STORE2   *STORE FILE COUNTER
256        MOVW   16@A0,#0   *UNLOCK RECORD
257        BRK    #0
258  *FLAG TO DUMP RECORD TO SCREEN
259        LDA    .A0,SVC BLOCK
260        MOVW   @A0,#9       *ID CODE
261        MOVW   6@A0,#90     *LENGTH OF LINE
262        MOVW   8@A0,#32       *SEND SPACE AT END OF ROUTINE
263        MOVL   10@A0,#INPUT1  *ADDRESS
264        BRK    #0
265  *PRINT A LABEL
266        LDA    .A0,SVC BLOCK
267        MOVW   @A0,#18      *IDENTIFYING FUNCTION CODE (PRCHAR)
268        LDA    .A5,INPUT1
269        STL    .A5,/STORE1
270        LDW    .D1,#25      *MAX # OF CHAR IN NAME LINE
271        CALL   PRINT A LABEL LOOP
272        LDA    .A5,INPUT2
273        STL    .A5,/STORE1
274        LDW    .D1,#25      *MAX # OF CHAR IN ADDR LINE
275        CALL   PRINT A LABEL LOOP
```

```
276   LDA    .A5,INPUT3
277   STL    .A5,/STORE1
278   LDW    .D1,#25   *MAX # OF CHAR IN TOWN LINE
279   CALL   PRINT A LABEL LOOP
280   LDA    .A5,INPUT4
281   STL    .A5,/STORE1
282   LDW    .D1,#2    *MAX # OF CHAR IN STATE LINE
283   CALL   PRINT A LABEL LOOP
284   LDA    .A5,INPUT5
285   STL    .A5,/STORE1
286   LDW    .D1,#8    *MAX # OF CHAR IN ZIP LINE
287   CALL   PRINT A LABEL LOOP
288   LDA    .A5,INPUT6
289   STL    .A5,/STORE1
290   LDW    .D1,#5    *MAX # OF CHAR IN CODE LINE
291   CALL   PRINT A LABEL LOOP
292   CALL   PRINT LINE FEED   *CUE UP FOR NEXT LABEL
293   CALL   PRINT LINE FEED
294   BR     EOF TEST
295 *LABEL PRINTING ROUTINE
296 PRINT A LABEL LOOP
297   LDL    .A5,/STORE1
298   MOVB   7@A0,@A5
299   MOVB   6@A0,#0   *INSURES A ZERO THERE
300   ADDW   .A5,#1    *INCREMENT
```

```
301          STL    .A5,/STORE1    *STORE AWAY NEXT CHAR.
302          LDB    .D5,7@A0
303          CMPB   .D5,#13   *TEST FOR CARRIAGE RETURN
304          BE     PRINT LINE FEED
305          BRK    #0
306          CMPW   .D1,#0         * DEC COUNTER
307          SUBW   .D1,#1
308          BE     PRINT LINE FEED
309          BR     PRINT A LABEL LOOP
310 PRINTER FINISHED
311          LDA    .A0,SVC BLOCK
312          MOVW   @A0,#133  *ID TO CLOSE ALL FILES
313          BRK    #0
314          BR     START
315 *SUBROUTINE TO PRINT LINE FEED CARRIAGE RETURN
316 PRINT LINE FEED
317          LDA    .A0,SVC BLOCK
318          MOVW   @A0,#18   *IDENTIFYING CODE-PRINT CHAR
319          MOVW   6@A0,#13  *DUMP A LINE FEED TO PRINTER
320          BRK    #0
321          RET
322 *
323 ***********************************************
324 *          SUBROUTINES                     *
325 ***********************************************
```

```
326  *
327  ********************************************
328  *      JUMP TO DOS READY MODE              *
329  *      ENTRY CONDITIONS: NONE              *
330  ********************************************
331  JUMPDOS
332          CALL    CLOSE   *MAKE SURE CLOSED
333          MOVW    @A0,#264 *FUNCTION CODE
334          BRK     #0
335  ********************************************
336  *      SCREEN ROUTINE ENTRY CONDITIONS:    *
337  *STORE1 POINTS TO ADDR. NEXT LINE TO DISPL*
338  ********************************************
339  SCREEN DISPLAY ROUTINE
340          LDA     .A0,SVC BLOCK
341          MOVW    @A0,#8  *FUNCTION CODE
342  DISPLAY A CHARACTER LOOP
343          LDL     .A3,/STORE1
344          MOVB    7@A0,@A3
345          MOVB    6@A0,#0     *INSURES A ZERO IS THERE
346          ADDW    .A3,#1      *INCREMENT
347          STL     .A3,/STORE1    *STORE AWAY ADDR NEXT CHAR
348          BRK     #0
349          LDL     .D0,#0
350          LDB     .D0,7@A0
```

125

```
351          CMPB    .D0,#5
352          BNE     DISPLAY A CHARACTER LOOP
353          RET
354  ********************************************
355  * CLEAR SCREEN: ENTRY CONDITIONS: NONE *
356  ********************************************
357  CLEAR SCREEN
358          LDA     .A0,SVC BLOCK
359          MOVW    @A0,#8       *FUNCTION CODE
360          MOVW    6@A0,#H'1B  *ASCII CODE TO CLS
361          BRK     #0
362          RET
363  ********************************************
364  *    DISPLAY ERROR NUMBER ON VIDEO    *
365  ********************************************
366  ERROR HANDLER
367          LDW     .A1,2@A0      *LOAD ERROR CODE INTO A1
368          MOVW    @A0,#39       *IDENTIFYING FUNCTION CODE
369          STW     .A1,6@A0     *LOAD ERROR CODE INTO SVC
370          BRK     #0   *EXECUTE ERROR NUMBER DSPLY
371          RET
372  ********************************************
373  * FILL DISK BUFFER AREA WITH SPACES *
374  ********************************************
375  CLEAR BUFFER
```

```
376        LDW    .D0,#90      *D0 IS COUNTER - 90 BYTE BUFFER
377        LDA    .A4,DISK BUFFER AREA1
378 FILL LOOP
379        MOVB   @A4,#32      *LOAD SPACE
380        ADDW   .A4,#1       *INCREMENT ADDRESS
381        CMPB   .D0,#0
382        DBE    .D0,FILL LOOP  *90 BYTES
383        RET
384 ****************************************
385 * OPEN - CREATE A FILE SUBROUTINE      *
386 ****************************************
387 OPEN   LDA    .A0,SVC BLOCK
388        MOVW   @A0,#40      *IDENTIFYING FUNCTION CODE
389        LDA    .A1,FILENAME
390        STL    .A1,6@A0
391        LDA    .A1,PARAMETER LIST
392        MOVB   3@A1,#1      *CREATE THE FILE ON DISK
393        STL    .A1,10@A0    *PUT ADDR OF PARAMETER LIST IN SVC
394        BRK    #0           *EXECUTE SUPERVISOR CALL
395        TESTW  2@A0         *TEST FOR AN ERROR
396        BNE    OPEN EXISTING FILE  *IF AN ERROR, THEN FILE EXISTS
397        LDW    .A1,14@A0       * GET FILE ID #
398        STW    .A1,/FILE IDENTIFICATION NUMBER   * STORE FILE ID
399        RET                 *RETURN
400 ****************************************
```

```
401   *        OPEN EXISTING FILE ROUTINE          *
402   ************************************************
403   OPEN EXISTING FILE
404            CALL    CLOSE
405            LDA     .A0,SVC BLOCK
406            MOVW    @A0,#40         *IDENTIFYING FUNCTION CODE
407            LDA     .A1,FILENAME
408            STL     .A1,6@A0
409            LDA     .A1,PARAMETER LIST
410            MOVB    3@A1,#0    *OPEN EXISTING FILE CODE
411            STL     .A1,10@A0    * PUT ADDRESS OF PARAMETER LIST IN SVC
412            BRK     #0
413            TESTW   2@A0            *CHECK FOR ERRORS
414            BNE     ERROR HANDLER       *IF ERROR JUMP TO ERROR HANDLER
415   *GET LAST USED RECORD # FROM RECORD #0
416            LDW     .A1,14@A0    *GETS FILE ID
417            STW     .A1,/FILE IDENTIFICATION NUMBER
418            MOVW    @A0,#35         *IDENTIFYING FUNCTION CODE (READ)
419   *
420            STW     .A1,6@A0        *PUT IN SVC BLOCK
421            LDA     .A1,DISK BUFFER AREA1
422            STL     .A1,8@A0        *PUTS BUFFER ADDRESS INTO SVC
423            MOVL    12@A0,#0        *GET RECORD #0 (1ST IN FILE)
424            MOVW    16@A0,#0        *UNLOCK RECORD
425            BRK     #0
```

128

```
426          LDL      .A1,/DISK BUFFER AREA1
427          STL      .A1,/NEXT RECORD NUMBER
428          RET
429   ********************************************
430   *          CLOSE THE FILE                  *
431   ********************************************
432   CLOSE
433          MOVW     @A0,#133     *CLOSEF CLOSE ALL FILES
434          BRK      #0      *CALL TO SVC CLOSE ALL FILES
435          RET
436   ********************************************
437   *    MESSAGE & WORKING STORAGE AREA        *
438   ********************************************
439   SVC BLOCK
440          RDATAB   32,0
441   STORE1 RDATAB   4,0
442   STORE2 RDATAB   4,0
443   FILENAME
444          TEXT     'LIST/DAT'
445          DATAB    13      *FILE NAME TERMINATOR
446          DATAB    0            *PRESERVE EVEN ADDRESS LOCATIONS
447   FILE IDENTIFICATION NUMBER
448          DATAW    0
449   PARAMETER LIST
450          DATAB    87,90,70,1,32,0
```

129

```
451 NEXT RECORD NUMBER
452     RDATAB  4,0      *STORES LAST USED RECORD #
453 KEYBOARD STORE1
454     DATAB   0        *STORAGE AREA FOR OPERATOR INPUT
455     DATAB   0        *KEEPS ADDRESSING ON EVEN NUMBER LOCATIONS
456 TEMPORARY KEYBOARD INPUT
457     RDATAB  2,0      *TEMPORARY KEYBOARD STORAGE
458 MENU OR ADD PROMPT
459     DATAB   13,13
460     TEXT    'Do you wish to <A>dd another name or <R>eturn to menu?'
461 MENU  TEXT  'MENU'
462     DATAB   13,13
463     TEXT    '============================================='
464     TEXT    '============================================='
465     DATAB   13,13,13,13
466     TEXT    '<A>  Add a name to the list '
467     DATAB   13,13
468     TEXT    '<S>  Search for a name'
469     DATAB   13,13
470     TEXT    '<M>  Modify a record'
471     DATAB   13,13
472     TEXT    '<P>  Printout address labels'
473     DATAB   13,13
474     TEXT    '<R>  Return to DOS Ready'
475     DATAB   13,13
```

```
476      TEXT    'Select the appropriate letter  > '
477      DATAB   5
478 ADD SCREEN DISPLAY
479      TEXT    'ADD A NAME TO LIST '
480      DATAB   13
481      TEXT    '-----------------------------------'
482      TEXT    '-----------------------------------'
483      DATAB   13,13
484      DATAB   5,0
485 SEARCH SCREEN DISPLAY
486      TEXT    'SEARCH FOR A NAME'
487      DATAB   13
488      TEXT    '-----------------------------------'
489      TEXT    '-----------------------------------'
490      DATAB   13,13
491      DATAB   5,0        *TERMINATOR CHARACTER
492 SEARCH PROMPT
493      TEXT    'Enter name to search for: '
494 SEARCH NAME INPUT
495      RDATAB  26,0
496 HIT ENTER PROMPT
497      TEXT    'Hit the <ENTER> key to return to menu '
498 MODIFY SCREEN DISPLAY
499      TEXT    'MODIFY A RECORD'
500      DATAB   13
```

```
501         TEXT    '-------------------------------------------'
502         TEXT    '-------------------------------------------'
503         DATAB   13,13
504         DATAB   5,0
505 PRINTOUT LABELS SCREEN DISPLAY
506         TEXT    'PRINTOUT ADDRESS LABELS'
507         DATAB   13
508         TEXT    '-------------------------------------------'
509         TEXT    '-------------------------------------------'
510         DATAB   5,0
511 *PROMPT  AND INPUT STORAGE AREA FOR "ADD" ROUTINE
512 PROMPT1  TEXT   'Enter name > '
513 PROMPT2  TEXT   'Enter street address   > '
514 PROMPT3  TEXT   'Enter city or town > '
515 PROMPT4  TEXT   'Enter 2 letter state abbreviation > '
516 PROMPT5  TEXT   'Enter zip code  > '
517 PROMPT6  TEXT   'Enter special code or comment > '
518 DISK BUFFER AREA
519 INPUT1   RDATAB  25,0    *STORES NAME
520 INPUT2   RDATAB  25,0    *STORES STREET
521 INPUT3   RDATAB  25,0    *STORES TOWN
522 INPUT4   RDATAB  2,0     *STORES STATE
523 INPUT5   RDATAB  8,0     *STORES ZIP
524 INPUT6   RDATAB  5,0     *STORES CODE
525          END     START
```

Chapter 11
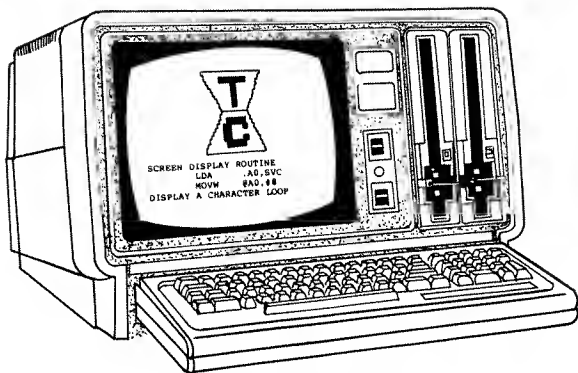
The TRS-80
Model II/16 Microcomputer

The Radio Shack Model II computer, shown in Fig. 11-1, has been popular for many years. It offers a lot of features that make it one of the most versatile microcomputers in its price range on the market.

It is possible for TRS-80 Model II computer owners to also enjoy the benefits of the MC68000 microprocessor. There are several ways in which current Model II users can utilize the equipment they already have and still take advantage of the newest technology.

One problem created by the rapid growth of the electronics industry is obsolescence. A small businessman may invest several thousand dollars in computer equipment, only to have it become "obsolete" in two or three years. Machines that can store more, run faster, and have better features seem to be released while their predecessors have only been on the market for a year or two. But we say that if a person purchases a microcomputer and sets it up to perform certain tasks, as long as it is capable of handling those jobs, it never becomes obsolete as far as he is concerned. It still does the work which the owner initially intended.

Even though production of the Model II has stopped, Radio Shack has not forgotten their customers who have previously purchased these machines from them. Thoughtful planning at the time the Model II was developed keeps it from becoming outdated too easily. We will explain two ways in which current Model II

Fig. 11-1. The TRS-80 Model II microcomputer. (Courtesy of Radio Shack)

owners can benefit from the MC68000 technology and still keep their machine.

The Model 16 is designed so that it can have two other video terminals connected to it. This will allow a total of three operators to simultaneously access the Model 16. The term given to a machine with this ability is "multi-user." These remote terminals or workstations allow each user to be working on different programs at the same time. One person may be doing payroll chores while another is running an accounts receivable program. The machine accesses every station so quickly that it appears to each person that the computer is dedicating all of its time to him alone. You will see this type of operation referred to as "multi-tasking" in the computer industry. As we have stated, it means that the computer can perform many tasks at once, each independent of the others.

So, the Model II owner who finds he has the need to establish a remote video terminal in his business, may wish to purchase a Model 16. The manufacturer has made the Model 16 compatible with the Model II in that the Model II can then be used as a remote workstation.

Furthermore, those Model II owners who need more comput-

134

ing power can upgrade their existing machine to incorporate the MC68000 microprocessor. This is done by placing a TRS-80 Model 16 Enhancement Option into the Model II. It consists of two circuit boards, shown in Fig. 11-2, and a power supply. One circuit board contains the MC68000 and its associated circuitry while the other houses more RAM memory chips. The existing Model II power supply is not sufficient to deliver all of the current needed to run the extra components so it must be replaced with a heavier duty unit.

The Z80 microprocessor remains in the Model II. Both the 16 and the II/16 put the Z80 to work handling input and output communications with peripherals as well as other "housekeeping" chores. This frees the MC68000 from doing many jobs so that it can dedicate itself to just processing the instructions in the program it is currently executing.

All of the assembly-language programs and subroutines listed
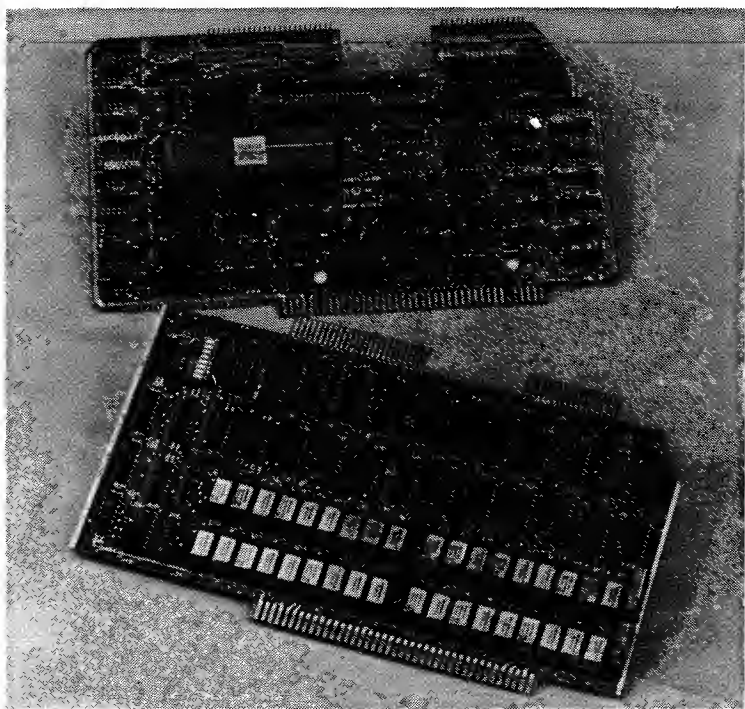
Fig. 11-2. The two circuit boards that comprise the Model II Enhancement modification. With the installation of these two boards and the upgrading of the existing Model II power supply, the popular Model II microcomputer can utilize the powerful MC68000 circuitry found in the Model 16. (Courtesy of Radio Shack)
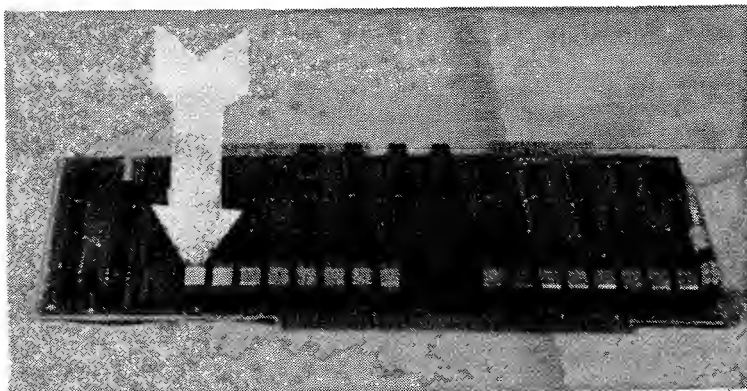
Fig. 11-3. The memory circuit board. The arrow points to the RAM integrated circuits. (Photograph by George M. Keen)

in this book will run on either a Model 16 or a Model II/16, that is, a Model II which has received the Enhancement Option. No changes whatsoever are necessary. They are completely compatible. The only time you would have to worry about differences between the two machines would be if your program accessed memory addresses above 256K. The Model 16 can have a maximum of 512K of RAM, but the modified Model II/16 has an upper limit of 256K.

Figure 11-3 points out the RAM chips. It is interesting to note that the RAM chips used are 64K by 1-bit each. In order to store one byte of information it requires 8 bits. Therefore 8 chips with one bit being stored in each are needed to comprise one byte of data. Chips are in banks of 64K but these are configured in 128K groups making the available RAM on a Model 16 128K, 256K, 384K, and 512K.

Another difference between the II/16 and the 16 is the disk drives. Since the disk drives are not changed when a Model II is upgraded, the Model 16 can store more information on a disk. So if you have a program which in some way needs to keep track of the amount of data to be stored on a disk, keep in mind that there is a difference in the storage capacity between a 16 and a II/16 disk.

The video character generator in the Model II is the same circuit board used in the new Model 16. Therefore, the entire ASCII code set is identical and programs which employ any of these codes need not be modified for Model 16 or II/16 applications.

Even though the fascinating world of electronics is making incredible technological breakthroughs, the purchaser of a Model 16 or II/16 does not have to fear the obsolescence of his computer in the near future. Radio Shack has wisely set up this machine so that it

can be easily converted to a completely functional 32-bit computer. It uses a 16-bit bus, but by using the process of multiplexing it can internally handle 32 bits in preparation for the day when 32 bit systems become the next generation of microcomputers. It appears as if the fine line that divides a "microcomputer" and a "minicomputer" is becoming more vague with each passing year. We are using equipment whose inner workings utilize circuitry that approaches the razor's edge of technology. These are exciting times for those of us who are involved in the field of computers.

Chapter 12
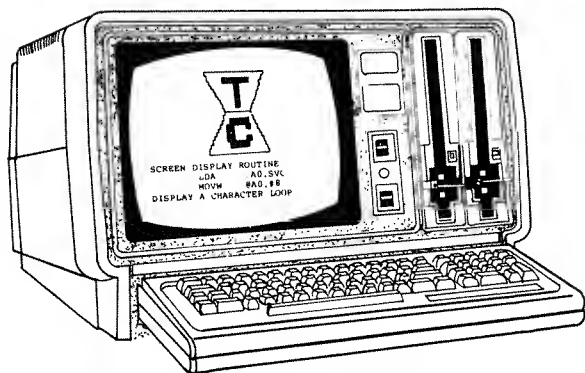
Interpreting Assembler Listings

The listing generated by the Assembler-16 can often help you "debug" your assembly-language programs, that is, find the logic and syntax errors. The Assembler-16 program allows us to print out a hard copy (paper listing) on a printer. This gives us a listing we can study more easily than the listing which appears on the screen during the assembling process.

In Chapter 3 we mentioned that the EQUate op code does not take up any memory space in the final machine-language program form. When we first used EQU in a program, we needed to place a decimal number 13 into the program to act as a terminator character. The ASCII code 13 is the command for a carriage return and a line feed. However, by using the EQUate instruction the machine was not actually storing that value at the memory location where the label was defined as we had incorrectly assumed. Therefore, our routine did not work. By examining a paper printout of the assembled listing, we quickly spotted the problem. The EQU op code is to be used when you need to let a word or phrase represent a value. At the time of assembling, the assembler replaces every occurrence of each label with the value that has been assigned with the EQU instruction. But the actual line where the EQU label is defined in the program does not result in any kind of storage at that point. So the following line did not work:

```
TERMINATOR
    EQU    13
```

138

In the above instruction, every occurrence of the label TER-MINATOR within the program is replaced by the numerical value 13. This line is strictly a programming tool which the assembler allows us to use. It does not take up any room in the final machine language form. It only plays a part during the assembling process.

Once we realized the problem, we searched for another mnemonic that would properly fill our requirements. After examining some possibilities, we felt the best choice was to use the DATA op code, which we did. DATA actually reserves memory space within the program and places the value you specify in that address. For example, DATAB 13 would reserve one byte of memory and store a decimal 13 in that location.

The point we wish to make here is that the assembler listing can be a great debugging tool. And when writing programs in this somewhat complicated language, every debugging aid available to us is most beneficial.

In order to get the Assembler-16 to generate a paper copy of an assembled program, we select the "P" or print option. The format to use is shown below.

```
ASM16 file name {P}
```

First the assembler must be loaded by typing ASM16. The file name is, of course, the name of the program you wish to list. The option to generate a printout is given last. The letter P must be surrounded by those funny shaped symbols { and }. These are called "braces." A space should be left between each parameter in the instruction line. For example, let us say we are going to assemble a mailing list program called MAILLIST. The proper format for assembling and printing the assembled listing would be:

```
ASM16 MAILLIST {P}
```

The following is a complete program that will clear the screen, move the cursor two lines down from the top of the video display and 31 character positions to the right, then finally return to the TRSDOS ready mode. This listing shows the entire assembler listing which we will discuss in detail.

```
1                           START
2    000000    4EBA   10            CALL      CLEAR  SCREEN
3              001A
                             POSITION CURSOR
4    000004    30BC   3             MOVW      @A0,#10
5    000008    317C                 MOVW      6@A0,#2
              0002
              0006
```

```
6      00000E    317C               MOVW      8@A0,#31
                 001F
                 0008
7      000014    4E40               BRK       #0
8      000016    30BC    8  JUMPDOS    MOVW      @A0,#264
                 0108
9      00001A    4340               BRK       #0
10                          CLEAR  SCREEN
11     00001C    41FA   16           LDA       .A0,SVC BLOCK
                 0010
12     000020    317C               MOVW      @A0,#8
                 0008
13     000024    317C               MOVW      6@A0,#H'1B
                 001B
14     00002A    4E40               BRK       #0
15     00002C    4E75               RET
16                          SVC  BLOCK
17     00002E      00   16           RDATAB    32,0
18     00004E  00000000+  1  *        END       START
```

We briefly touched on assembler listings in Chapter 2. As stated there, the first column of numbers on the left-hand side of the listing are the line numbers of the source code. Line numbers are in decimal form. This is the way the line numbers would appear if you were to go back and load the editor and the specified source code. These numbers may not be the way you left them upon exiting from the editor program, since the editor does not give you line numbers when you first begin typing nor when you insert new lines into a program.

Should you ever have the need to see each line of instruction with a line number beside it while you are working on a source code, simply save the source code, type DE ALL to delete all of the text in memory, and finally enter CO file name, where "file name" represents the name of the current file on which you are working. CO is the command concatenate by loading a source file from disk into the computer's memory. The thinking behind the symbol "CO" is for the function of concatenating, since there may be a time when a programmer wishes to "concatenate" or add another program from a disk onto the source file currently in memory. Anytime you need to edit several lines of a source code, it is very helpful to have line numbers.

Appearing immediately to the right of the line number will either be a blank space or a letter "A" through "I." Usually your listing will have a blank space indicating this line is from the primary source file. A character means that line has been "copied" or loaded into the program lines from another file on the disk. The COPY pseudo-op code allows us to perform this operation. The character in the assembly listing is an "A" when it is the first file copied into

the program, "B" when it is the second, "C" the third, and so on up to and including "I." The limit is nine, "I" being the ninth letter of the alphabet.

After the line number appears a six-digit figure disclosing the relative address location of that particular instruction. There is a slight shroud of mystery concerning the actual addresses where a program is to reside. The Model 16 contains no ROM circuitry with a language interpreter in it. All of its memory is RAM. TRSDOS loads into RAM starting at the beginning of the machine's memory. It takes up about 48K of the computer's RAM memory. At the time of powering up the machine any other interpreters, systems, or debugging programs (such as Debug) are loaded. By the time our program is loaded into memory, its starting address is a fairly high number in memory. But at no time are we told the exact RAM location of our program. The starting address of our program, wherever that may be, is given a relative address by the assembler. The first instruction falls on relative address zero. Each instruction is then referenced to that address.

In our sample listing above, note that the first instruction, CALL CLEAR SCREEN, is given the address of zero. The next actual instruction, MOVW @ A0, #10, is shown to have a relative address of 000004. Ergo, the first instruction requires four bytes of memory, taking up locations 000000, 000001, 000002, and 000003.

The labels START and POSITION CURSOR do not take up any memory space, as you will notice. As we have noted before, labels represent memory locations, but do not take up space themselves. START identifies the first instruction of the program, located at 000000. POSITION CURSOR points to the first address of the four-byte long MOVW instruction, which has a relative address 000004.

The third column group of numbers shows the actual hexadecimal code which will be converted to binary for the microprocessor to utilize. Should an instruction require several bytes of memory to store, the additional code is listed directly beneath the first part of the instruction. Let us examine line four in our sample listing. The instruction reads:

```
30BC          MOVW       @A0,#10
000A
```

In this case we see the instruction for moving an immediate word (two byte) value indirectly into register A0. The generated code is 30BC. The value to be moved is decimal ten. The extension

of that instruction is found on the next line in the same column positions, 000A. As you know, 000A is the hexadecimal equivalent of a decimal ten.

You may note that the few examples we discussed in Chapter 3 show the hexadecimal instruction codes to be eight digits long. Here you see instructions are grouped in four column positions, comprising two bytes as opposed to four. This is because in this particular sample listing all instructions act on a length of two bytes, namely one word. In Chapter 3 the example given acted upon long words (with the exception of EQUate) and therefore required eight character positions to depict.

```
25    0000062   4D454E55   25    MENU   TEXT   'MENU'
26              0000000D                 EQUW   13
27    0000066   3D3D3D3D                 TEXT   '===='
```

Eight column positions from 10 to 17 are reserved for the display of the assembled code. If only four positions are needed, then the first four columns starting at the left going toward the right remain blank.

In the next column, adjacent to the instruction code, is the relocatable indicator. It can only be seen once in our program and that is in the last statement, line 18. In a moment we will discuss that further.

The next set of numbers in the assembler listing shows the line number of the source code where a label is first defined. If no label is present, these column positions 19 through 22 remain blank spaces. Again referring back to the sample program, examine line number 11.

```
11    00001C       41FA    16           LDA      .A0,SVC BLOCK
```

In the operand, the label SVC block is used. This label is defined on line 16 of the source code. Thus the number 16 is displayed in the proper columns.

Since we are allowed to use labels up to 45 characters in length, it is often necessary to place a label on a line prior to the actual line where the instructions are located. It is also necessary to dedicate an entire line to a label if that label is to be used as a "global" reference, that is, it can be referred to from any spot in the program; it is not limited to any particular range.

In both of these cases, the label is actually assigned to the set of instructions which start on the following line. This can be seen in line one and line three of our program:

```
1                              START
2  000000   4EBA   10               CALL      CLEAR SCREEN
            001A
3                              POSITION CURSOR
4  000004   30BC   3                MOVW      @A0,#10
```

If a label is not a part of the original source code but has been copied into it by using the COPY directive as we just discussed, a letter will be produced and shown back to back against the "line number defined" column. As we have previously stated, with the appearance of a letter in column five, the file it came from is defined as "A" for the first file, "B" for the second, and so forth in keeping with this assembler's standards. Since we do not have any copying taking place in our sample program, column 23 remains a blank space.

Column 24 will contain either a blank space or an asterisk (*) symbol. When the assembler must branch to another section of the program or make a reference to a label, this column indicates the direction in which the label can be found. Labels which have been defined prior to executing this line will produce an asterisk here. This can be seen on the last line of our program, line number 18.

```
18   00004E   00000000000+  1  *          END       START
```

The symbol START is a backward reference, that is, it was established before reaching this location in the program.

A blank space shows that the reference is further on in the program, said to be a forward reference. All of the other occurrences in our program here are forward references, thus, column 24 contains a space in all of these.

Several things found in line 18 should also be discussed. As we have said many times before, all addresses in the Model 16 are four bytes long. The operand of the END statement in line 18 as START, which defines the entry point location where the computer will begin execution once the program has been loaded into memory. This explains why there are eight zeros in columns 10-17.

Finally, the plus sign (+) immediately following the hexadecimal address shows that the symbol is relocatable. This means that the linker program (LINK16) can alter addresses of code or information. A blank space indicates it cannot be moved. A period means it is an external signal which is a special case and is beyond the scope of the material the average programmer needs to know in order to begin writing assembly-language programs for the Model 16.

The remainder of all information to the right of the backward/forward reference marker in the listing is the source code itself as we typed it in. This particular sample program does not show any comments but the comment section is printed at the extreme right of the listing. If a comment is too long to fit on the paper, the assembler will simply print the rest of the characters on the following line.

We have already shown how the EQU op code does not take up any memory space as an assembled listing discloses. Other instructions in a program, such as branches, can also be followed by close examination of the hexadecimal codes generated by the assembler. Jumps, calls, and other types of branching commands can be seen in the listing. Let's look at the CALL statement in line two of our example:

```
2    00000000    4EBA    10        CALL     CLEAR SCREEN
                  001A
```

The generated code for CALL is 4EBA. Note the instruction's extension code 001A. This is a decimal 26. The CALL op code causes the computer to jump to another memory address and begin executing instructions from that point. It does this by adding the number of bytes of displacement to the number which is currently in the PC (program counter) register. Displacement may be in either direction, forward or backward in the memory in respect to the current address. Here, the machine is directed to go to the section of the program defined by the label CLEAR SCREEN where a routine to blank out the video display is located. Notice the address at which the CLEAR SCREEN routine is located.

```
10                            CLEAR SCREEN
11   00001C    41FA    16         LDA      .A0,SVC BLOCK
```

The actual set of instructions for CLEAR SCREEN begin on line 11. The relative address is shown as being 00001C. This is a decimal 28. So in order for the computer to execute the instruction in line 11 next, it must be told how many bytes ahead to travel. As we previously mentioned, the extention code for CALL is 001A hexadecimal, or 26. The displacement, therefore, is 26 bytes. This is the value to be added to the number already residing in the program counter register. The CALL instruction itself takes up two bytes. The addition of 2 bytes for the CALL instruction plus 26 bytes for the displacement equals 28. As you will note the location of the CLEAR SCREEN routine is indeed at the relative address of 28 decimal, 001C hexadecimal.

144
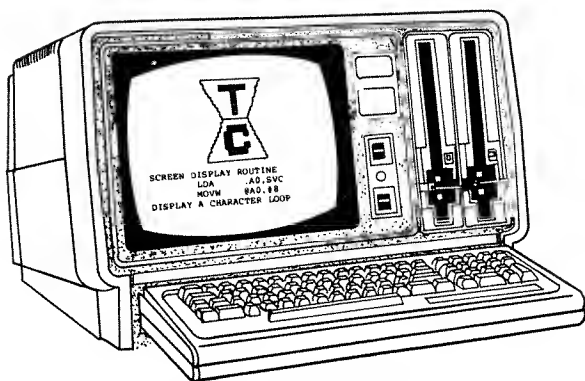
Chapter 13

Some Programming Techniques

In this chapter we will go further into the world of assembly language by establishing some simple routines which the reader can use when he needs to perform certain tasks. We will also cover some ideas to help you debug your machine-language programs.

The listings of the concepts discussed are kept as short and clear as possible. In this way they are easier to follow and to build into other programs.

TIMING LOOPS ANO LOOPS IN GENERAL

The need to create loops is a very common occurrence in assembly language. We cannot think of any program which does not use at least one simple loop. Here we will show how to create a loop similar to a FOR/NEXT loop in the BASIC language.

Before entering a loop, it is customary to establish the count by placing the desired value in a memory location or perhaps in a register. The sequence of instructions which lies between the beginning and the end of the loop can be repeated any number of times. If no processing takes place other than incrementing (or decrementing) a number (a counter) and testing to see if the count has reached a particular value, the loop can be said to be used as a time delay. The machine simply sits and counts to itself for a predetermined length of time.

By the addition of the NOP instruction, the time can be precisely controlled. The mnemonic NOP represents "no operation."

An NOP will simply cause the computer to waste a few clock cycles of time. Condition bits in the status register are not changed. The address stored in the PC (program counter) register is advanced to point to the location of the next instruction.

Whether we are setting up a time delay or a loop to repeat a series of instructions, we need the same ingredients. A number can be placed in a register and it can be incremented or decremented until it reaches the desired value.

Depending upon the type of assembler, a loop listing for the Z80 microprocessor might look something like this:

```
          LD    B,0FFH
LOOP      DJNZ  LOOP
```

The same function can be performed in a similar manner on the MC68000 by the following:

```
          LDW   .D0,#H'FF
LOOP      DBE   .D0,LOOP
```

In the above routine, the hexadecimal number FF (decimal 255) is loaded into data register D0. The next line does all the work. First, the value in D0 is decremented, that is, one (1) is subtracted from its current value. The Z flag is checked to see if it has been set or cleared. If the value in D0 is not equal to zero, execution branches back to the same line where again D0 is decremented and tested again.

The letters chosen to represent that op code at first seemed backward to us. We tend to want to think of the instruction DBE as meaning "branch if equal" when in fact it is branching when the value in D0 is not equal to zero. But keep in mind that what is actually happening here is the testing of the Z flag in the status register. The Z bit is checked for a set or cleared condition. The Z bit is clear (it contains a zero) as long as the value in D0 (in our example) is not equal to zero. When the register has reached zero, the flag is set (it contains a value of one). So a little thought is needed when setting up a loop on the Model 16.

When the routine above is run on the Model 16, it executes far too rapidly to be seen.

Should we want to extend the count it makes a rather lengthy routine for the Z80 since its registers cannot hold a value larger than FF. Registers must be paired together if numbers larger than 255 are required. It is no problem accomplishing a count of, let's say, hexadecimal FFFF (65,535 decimal) when programming the MC68000 since the register can directly store a two-byte value.

```
          LDW     .D0,#H'FFFF
   TIMING LOOP
          DBE     .D0,TIMING LOOP
```

INCREMENTING THE VALUE STORED IN AN ADDRESS

Often it is necessary to take a value which is stored in a memory location and add a value to it. The routine we show here uses the ADD mnemonic to increment the value. By using this instruction we are not limited to incrementing by only one, rather, it can be increased by any amount.

Let's assume that we want to increase the current value stored in the memory location whose label is STORE1. In our example we will increment the value by one.

The first thing to do is to load the contents of STORE1 into an address register where we can use the ADD instruction. As you recall, a slash (/) tells the computer "the contents of" that address.

Next we add the desired value to the register. And finally, the STore command will place the new number back into the address STORE1.

Below is a listing of our sample routine. Address register A1 has been chosen to perform the addition. We avoid using A0 since many times that register is tied up in a program keeping track of the SVC block. Also, we used the suffix "L" so that this routine could be used in any application where the number goes as high as FFFFFFFF.

```
          LDA     .A1,/STORE1
          ADDL    .A1,#1
          STL     .A1,/STORE1
```

TESTING FOR GREATER THAN AND LESS THAN

In Chapter 6 we showed how the CMP (compare) instruction is used to set the various bits in the status register.

Usually we test the Z flag bit to check for an "equal to" condition.

But what if we wanted to see whether a value was greater or less than a particular number? There are other bits which can be examined.

As we pointed out in Chapter 6, the CMP code performs a subtraction operation. The various bits in the status register are either set or cleared depending upon the results of the subtraction. The result of this computation is not stored anywhere. Its purpose is solely to set the bits in the status register.

Perhaps a sample problem can best demonstrate how we can

test for a greater than or less than condition. Assume a number has been placed in data register D0 and we want to find out if the value stored there is greater or less than, say, the decimal number 100. This could be done with the following instruction:

```
CMPW      .D0,#100
```

This instruction will cause the machine to subtract 100 from D0. If the Z flag equals a one, then the two numbers are the same. Should the Z bit contain a zero, the value in D0 is not 100. So we can use the Z bit to find out an "equal to" or "not equal to" condition.

Another bit, the "V" or "borrow" flag, is set if D0 is less than 100 and the computer was forced to borrow to complete its calculation.

There are only three possible combinations of equality. Either one number is equal to, less than, or greater than another number. So all we really have to do is test for equal and either greater or less than. First test for equality and then test for a less than condition. If execution falls through to the next program instruction, then it must be greater than.

Next we put the BRanch instruction to work. There are over a dozen variations of the BRanch command. Each one tests a different bit of the status register and either branches or does not branch to another location in the program. The variation to use will depend upon the particular circumstance and how the program flow is to proceed when certain conditions are met.

In alphabetical order, the available BRanch commands are:

```
BC    branch if carry
BE    branch if equal (referring to Z flag)
BGE   branch if greater than or equal
BGT   branch if greater than
BH    branch if higher than
BHS   branch if high, same
BLE   branch if less than or equal
BLO   branch if low
BLT   branch if less than
BN    branch if negative
BNC   branch if no carry
BNE   branch if not equal
BNH   branch if not higher than
BNV   branch if no overflow
BP    branch if positive
BR    branch - always branch regardless of status register
BV    branch if overflow
```

DEBUGGING IDEAS

Writing assembly-language programs with the Assembler-16 package is a very time-consuming task. If you just want to make one

tiny change in the source program, it is necessary to run the editor program, load your source code, make the change, save the new version, quit the editor, run the assembler and assemble the source code, run the linker program, then finally, run your program! Hopefully the change you made fixed the particular problem you were having. If not, well, back to the drawing board. It does take a good four to five minutes to go from the source program to an actual test running of the program. We would like to mention a few ideas which we found useful in debugging our programs and which helped to speed up program development.

As we explained in Chapter 9, an odd address trap error is caused when the computer tries to execute an instruction which falls on an uneven memory address location. The line at which the error occurs in the program may not necessarily be the line where the offending problem is located. For you BASIC programmers, take a look at this erroneous program:

```
10   S = 0
30   PRINT X
40   S = S+X
50   NEXT X
```

Here, the programmer forgot to type in line 20 which should read something like

```
20   FOR X = 1 TO 10
```

By leaving that line out, the computer will generate an error in line 50 (NEXT WITHOUT A FOR). But that line is the symptom and not the cause.

Similarly in MC68000 programming, an odd address trap error could be caused by the use of an uneven number of bytes at some other point in the program.

To help locate the cause of an odd address trap error, assemble the program and get a listing on paper. The second set of numbers on the listing is the relative address locations. Search for an odd number in the least significant digit location. Acceptable even digits are of course 0, 2, 4, 6, 8, A, C, and E.

```
1   000000   41FA   43   OPEN   LDA    .A0,SVC BLOCK
               009A
2   000004   30BC        MOVW   @A0,#40
               0028
3   000008   43FA   45          LDA    .A1,FILE NAME
```

```
              00B2
 4   00000C   2149          STL    .A1,6@A0
              0006
 5   000010   43FA  49      LDA    .A1,PARAM LIST
              00B4
```

In the above sample piece of an assembler listing, we can quickly look at the last digit in the relative address column to check for any occurrences of an odd address. Everything looks all right in this listing, the least significant numbers being 0, 4, 8, C, and 0.

You should be careful when you define a byte of memory space in a program. This can cause a later instruction to fall on an odd address. You can use all the EQUB instructions you want, since equating a byte does not take up any memory space. However the command DATAB does set aside a byte of memory and can give trouble. Perhaps it is a good idea to define two bytes instead of one when possible even if the other byte is not used. It will act as a place holder and keep the relative address locations even.

Another trick you may need to help in the process of debugging is the ability to print a numerical value on the screen. This is not as easy as it sounds at first. Let's say that you are working on a program that is designed to open a file and place some information into it. Usually you will test the routines by placing some "phoney" information into the first few records. If a file can be opened and data written to the first few records, then in all likelihood hundreds of records of information can be written to it. There may be a time when you need to know what the record number is of the record currently being written or read from the file. Assume register A1 is currently holding the number of the next record to be written to the disk. We may want to introduce a "flag" into the program. In other words, we need to be able to print that record number on the video screen so we can examine it.

Why can't we just place that number into the SVC block and call the display a character routine? This would not display the actual number, but rather the ASCII function of that value. As an example, suppose the record number stored in A1 was a one. Putting a one into the display a character routine would send the control code "1" to the screen, which is the command to turn on the blinking cursor.

The ASCII codes for the numbers 0 through 9 begin with 48 (decimal) and continue through to 57. Therefore, by adding the number 48 to the number stored in A1, we can get the supervisor routine to display the actual decimal number. The drawback to this is that it will not handle any number greater than 9. The advantage of this trick is that it enables you to quickly and without too many

steps, write a routine that will help you see what the machine is doing. If A1 holds a number between 0 and 9, you print that value on the screen with this set of instructions:

```
LDA     A0,SVC BLOCK
ADDW    .A1,#48      *ADD 48 TO THE VALUE IN A1
MOVW    @A0,#8       *IDENTIFYING FUNCTION CODE
STW     .A1,6@A0     *PUT VALUE INTO BLOCK
BRK     #0           *EXECUTE SUPERVISOR CALL
```

A WORD ABOUT "END"

Another simple but very important instruction is END. We have mentioned earlier in this book that we found that if the END statement followed by a label showing where the first instruction of the program is to begin execution is left out, the program would not work. Also, it is easy to forget to stick the END instruction in a program. The Assembler-16 manual never shows it in any of the supervisor routine listings because they are not intended to be whole programs, so it is easy to forget that statement. The bad news is that the assembler does not pick up this error. If you omit the END statement, no error message nor warning is generated by the assembler. Even worse is the fact that your program will in most cases just do nothing. When you try to run your program the screen will merely come back with TRSDOS ready. So you don't know whether anything has actually happened. You could spend hours wondering why a program did not work.

We want to stress that you should carefully inspect any program that does not appear to do anything, checking for the proper label at the top of the program (such as BEGIN or START) and the correct termination instruction (END   START,   END BEGIN, or the like). While this is a simple program, detecting this error is not so easy since the computer does not help you in detecting the omission.

Another problem we ran into was mentioned in Chapter 2. Occasionally we would get assembler errors by having a comment on the same line as an instruction. Problems have never occurred when an entire line was dedicated to a comment, beginning with an asterisk in the label column. In a few instances the assembler gave errors when a comment was listed on the same line, especially if it did not begin in the comment column of the source code listing, even though it may be preceded by an asterisk.

So, if you get an assembler error message on an instruction line that looks all right to you, remove the comment at the end of the line if one is present. This may clear up the error.

Appendix A

SVC Block Setup

The following charts show the function and position of all critical bytes in the supervisor block that are needed before you can call each of the supervisor routines developed in this book. Information that is placed within the block by the disk operating system upon return from execution of a routine is also shown.

These diagrams will help to give you a quick overview of the SVC block setup. For specific information as to the action of certain values placed in key positions, refer to the chapter in the text dealing with that particular supervisor call.

JUMP TO TRSDOS ROUTINE

Byte position

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	A	X	X	0	0																										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Byte-offset

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine
A - identifying supervisor code, 264 decimal

Upon Return from the Routine
X - TRSDOS error code (zero if no error occurred)

CLEAR THE SCREEN ROUTINE

Byte position

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	A	X	X	0	0	B	B	C	C																						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Byte-offset

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine
A - identifying supervisor code, 7 decimal
B - determines the size of printed characters, 40 or 80 characters
    per horizontal line
C - determines normal or inverse video printing

Upon Return from the Routine
X - TRSDOS error code (zero if no error occurred)

DISPLAY A CHARACTER ROUTINE

Byte position

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	A	X	X	0	0	B	B																								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Byte-offset

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine
A - identifying supervisor code, 8 decimal
B - the code which represents the desired character or control code
    that will be output to the screen

Upon Return from the Routine
X - TRSDOS error code (zero if no error occurred)

POSITION THE CURSOR ROUTINE

Byte position

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	A	X	X	0	0	B	B	C	C																						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Byte-offset

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine
A - identifying supervisor code, 10 decimal
B - lines down from the top of the screen
C - character positions from the left side of the screen

Upon Return from the Routine
X - TRSDOS error code (zero if no error occurred)

DISPLAY A LINE ROUTINE

Byte position

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	A	X	X	0	0	B	B	C	C	D	D	D	D	Y	Y	Z	Z														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Byte-offset

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine
A - identifying supervisor code, 9 decimal
B - size of the buffer
C - terminator character
D - memory address of the buffer

Upon Return from the Routine
X - TRSDOS error code (zero if no error occurred)
Y - if an error occurred, the number of characters not sent to the screen

154

Z - if an error occurred, the character that caused it

KEYBOARD CHARACTER ROUTINE

Byte position

```
 1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
 A  A  X  X  0  0  B  B  Z  Z
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Byte-offset         ↖  ↖ Y on exit
                   Y on exit
```

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine
A - identifying supervisor code, 4 decimal
B - determines wait or not wait condition

Upon Return from the Routine
X - TRSDOS error code (zero if no error occurred)
Y - indicates a valid ASCII code has been entered by user
Z - returned character from the keyboard

KEYBOARD LINE ROUTINE

Byte position

```
 1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
 A  A  X  X  0  0  B  B  C  C  C  C  Y  Y  Z  Z
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Byte-offset
```

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine
A - identifying supervisor code, 5 decimal
B - maximum characters allowed to be entered
C - address to store input line

Upon Return from the Routine
X - TRSDOS error code (zero if no error occurred)

Y - number of characters entered by operator
Z - indicates if carriage return was used to end input

CLEAR THE TYPE AHEAD BUFFER ROUTINE

Byte position

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	A	X	X	0	0																										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Byte-offset

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine
A - identifying supervisor code, 1 decimal

Upon Return from the Routine
X - TRSDOS error code (zero if no error occurred)

DISPLAY MESSAGE AND KEYBOARD LINE INPUT ROUTINE

Byte position

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	A	X	X	0	0	B	B	C	C	D	D	D	D	E	E	E	E	Y	Y	Z	Z										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Byte-offset

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine
A - identifying supervisor code, 12 decimal
B - length of message
C - length of keyboard input field
D - address to store keyboard response
E - address of text to be sent to the screen

Upon Return from the Routine
X - TRSDOS error code (zero if no error occurred)
Y - if an error occurs, the number of characters not sent
Z - if an error occurs, the character that caused it

PRINT A CHARACTER ROUTINE

Byte position

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	A	X	X	0	0	B	B																								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16	15	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Byte-offset

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine
A - identifying supervisor code, 18 decimal
B - character to direct to the printer

Upon Return from the Routine
X - TRSDOS error code (zero if no error occurred)

PRINT A LINE ROUTINE

Byte position

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	A	X	X	0	0	B	B	C	C	D	D	D	D																		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Byte-offset

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine
A - identifying supervisor code, 19 decimal
B - number of characters in the line
C - character to send at end of buffer to terminate
D - address of message (buffer address) to be printed

Upon Return from the Routine
X - TRSDOS error code (zero if no error occurred)

DISPLAY TRSDOS ERROR NUMBER

Byte position

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	A	X	X	0	0	B	B																								

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Byte-offset

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine
A - identifying supervisor code, 39 decimal
B - place the error number here

Upon Return from the Routine
X - TRSDOS error code (zero if no error occurred)

DISPLAY ERROR MESSAGE

Byte position

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	A	X	X	0	0	B	B	C	C	C	C																				

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Byte-offset

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine
A - identifying supervisor code, 52 decimal
B - place the error number here
C - place the address of the message storage area here

Upon Return from the Routine
X - TRSDOS error code (zero if no error occurred)

158

OPEN A DISK FILE

Byte position

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	A	X	X	0	0	B	B	B	B	C	C	C	C	Y	Y																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Byte-offset

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine

A - identifying supervisor code, 40 decimal
B - address where the disk file name is located
C - address of OPEN specifications, that is, codes to indicated record length, read or write status, attributes, and the like

Upon Return from the Routine

X - TRSDOS error code (zero if no error occurred)
Y - contains a special number created by TRSDOS for its internal use. This number must be used when referring to this file such as reading and writing to and from the disk

CLOSE A DISK FILE

Byte position

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	A	X	X	0	0	B	B																								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Byte-offset

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine

A - identifying supervisor code, 42 decimal
B - place here the special number which was returned by TRSDOS at the time the file was created

Upon Return from the Routine

X - TRSDOS error code (zero if no error occurred)

GET A RECORD FROM A DISK FILE

Byte position

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	A	X	X	0	0	B	B	C	C	C	C	D	D	D	D	E															

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Byte-offset

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine
A - identifying supervisor code, 35 decimal
B - place here the special number which was returned by TRSDOS at the time the file was created
C - address where the record information is to be placed once it is read from the disk
D - the number of the desired record
E - a non-zero value here locks the record

Upon Return from the Routine
X - TRSDOS error code (zero if no error occurred)

PUT A RECORD INTO A DISK FILE

Byte position

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	A	X	X	0	0	B	B	C	C	C	C	D	D	D	D																

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Byte-offset

Letters represent byte locations within the block in which values must be placed before calling the routine.

Byte-offset four and five must have a value of zero placed in them before calling any supervisor routine.

Before Calling the Routine
A - identifying supervisor code, 44 decimal
B - place here the special number which was returned by TRSDOS at the time the file was created
C - address where the record information to be placed on the disk is located

160

D - the record number where this information is to be placed within the disk file

Upon Return from the Routine
X - TRSDOS error code (zero if no error occurred)

Appendix B

EDIT16 Op Code Mnemonics

The following op code mnemonics, given in alphabetical order, are
used by the Assembler-16 program.

ADD	BRK	DEC	RET	SETF
ADDC	BRKV	DBLO	RTR	SETLO
ADDD	CALL	DBHS	ROL	SETHS
AND	CHK	DIV	ROR	SHL
BE	CLR	DIVU	ROLC	SHR
BNE	CMP	EXT	RORC	SHLA
BC	DBR	LD	SET	SHRA
BNC	DBE	LDA	SETE	ST
BP	DBNE	LDM	SETNE	STM
BN	DBC	LDP	SETC	STP
BV	DBNC	LINK	SETNC	SUB
BNV	DBP	MOV	SETP	SUBC
BGT	DBN	MUL	SETN	SUBD
BGE	DBV	MULU	SETV	TEST
BLT	DBNV	NEG	SETNV	TESTSET
BLE	DBGT	NEGC	SETGT	TEST1
BH	DBGE	NEGD	SETGE	TESTCLR1
BNH	DBLT	NOP	SETLT	TESTNOT1
BLO	DBLE	NOT	SETLE	UNLK
BHS	DBH	OR	SETH	XCH
BR	DBNH	PUSHA	SETNH	XOR

Appendix C

Assembler-16
Directives (Pseudo Op Codes)

ASECT	DSECT	PAGE	RSECT
COPY	END	RDATA	TEXT
DATA	EQU	REF	TEXTC
DEF	ORG	RES	TITLE

Appendix D
Decimal To
Hexadecimal Conversions

Only values from 0 to 255 are given, as 255 represents the highest number that can be stored in one byte.

Decimal	Hexadecimal	Decimal	Hexadecimal
0	00	19	13
1	01	20	14
2	02	21	15
3	03	22	16
4	04	23	17
5	05	24	18
6	06	25	19
7	07	26	1A
8	08	27	1B
9	09	28	1C
10	0A	29	1D
11	0B	30	1E
12	0C	31	1F
13	0D	32	20
14	0E	33	21
15	0F	34	22
16	10	35	23
17	11	36	24
18	12	37	25

Decimal	Hexadecimal	Decimal	Hexadecimal
38	26	79	4F
39	27	80	50
40	28	81	51
41	29	82	52
42	2A	83	53
43	2B	84	54
44	2C	85	55
45	2D	86	56
46	2E	87	57
47	2F	88	58
48	30	89	59
49	31	90	5A
50	32	91	5B
51	33	92	5C
52	34	93	5D
53	35	94	5E
54	36	95	5F
55	37	96	60
56	38	97	61
57	39	98	62
58	3A	99	63
59	3B	100	64
60	3C	101	65
61	3D	102	66
62	3E	103	67
63	3F	104	68
64	40	105	69
65	41	106	6A
66	42	107	6B
67	43	108	6C
68	44	109	6D
69	45	110	6E
70	46	111	6F
71	47	112	70
72	48	113	71
73	49	114	72
74	4A	115	73
75	4B	116	74
76	4C	117	75
77	4D	118	76
78	4E	119	77

Decimal	Hexadecimal	Decimal	Hexadecimal
120	78	161	A1
121	79	162	A2
122	7A	163	A3
123	7B	164	A4
124	7C	165	A5
125	7D	166	A6
126	7E	167	A7
127	7F	168	A8
128	80	169	A9
129	81	170	AA
130	82	171	AB
131	83	172	AC
132	84	173	AD
133	85	174	AE
134	86	175	AF
135	87	176	B0
136	88	177	B1
137	89	178	B2
138	8A	179	B3
139	8B	180	B4
140	8C	181	B5
141	8D	182	B6
142	8E	183	B7
143	8F	184	B8
144	90	185	B9
145	91	186	BA
146	92	187	BB
147	93	188	BC
148	94	189	BD
149	95	190	BE
150	96	191	BF
151	97	192	C0
152	98	193	C1
153	99	194	C2
154	9A	195	C3
155	9B	196	C4
156	9C	197	C5
157	9D	198	C6
158	9E	199	C7
159	9F	200	C8
160	A0	201	C9

| Decimal	Hexadecimal	Decimal	Hexadecimal
202	CA	230	E6
203	CB	231	E7
204	CC	232	E8
205	CD	233	E9
206	CE	234	EA
207	CF	235	EB
208	D0	236	EC
209	D1	237	ED
210	D2	238	EE
211	D3	239	EF
212	D4	240	F0
213	D5	241	F1
214	D6	242	F2
215	D7	243	F3
216	D8	244	F4
217	D9	245	F5
218	DA	246	F6
219	DB	247	F7
220	DC	248	F8
221	DD	249	F9
222	DE	250	FA
223	DF	251	FB
224	E0	252	FC
225	E1	253	FD
226	E2	254	FE
227	E3	255	FF
228	E4	…	..
229	E5	…	..

Appendix E
MC68000 Data Sheets

The following material was supplied courtesy Motorola, Inc. The MC68000 pin assignment, register format, and Motorola's mnemonic instruction set is given.

16-BIT MICROPROCESSING UNIT

Advances in semiconductor technology have provided the capability to place on a single silicon chip a microprocessor at least an order of magnitude higher in performance and circuit complexity than has been previously available. The MC68000 is the first of a family of such VLSI microprocessors from Motorola. It combines state-of-the-art technology and advanced circuit design techniques with computer sciences to achieve an architecturally advanced 16-bit microprocessor.

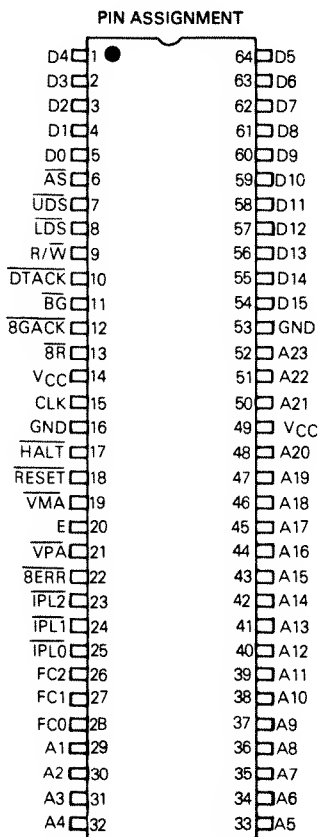The resources available to the MC68000 user consist of the following:

- 32-Bit Data and Address Registers
- 16 Megabyte Direct Addressing Range
- 56 Powerful Instruction Types
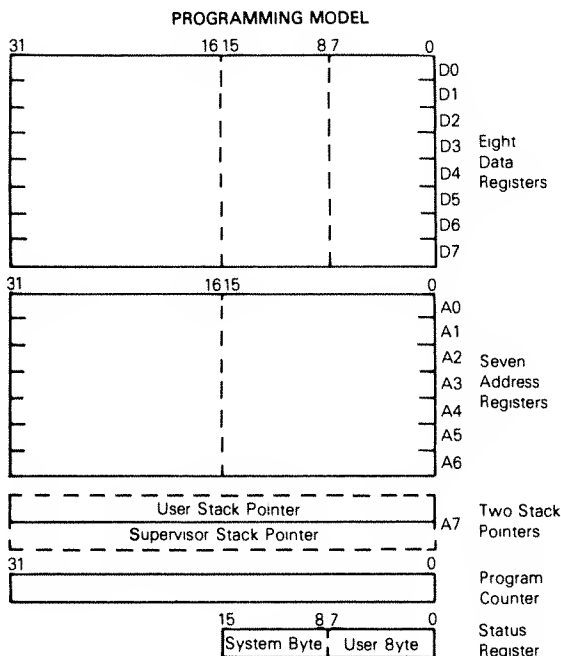
PROGRAMMING MODEL

170

- Operations on Five Main Data Types
- Memory Mapped I/O
- 14 Addressing Modes

As shown in the programming model, the MC68000 offers seventeen 32-bit registers in addition to the 32-bit program counter and a 16-bit status register. The first eight registers (D0-D7) are used as data registers for byte (8-bit), word (16-bit), and long word (32-bit) data operations. The second set of seven registers (A0-A6) and the system stack pointer may be used as software stack pointers and base address registers. In addition, these registers may be used for word and long word address operations. All seventeen registers may be used as index registers.

PIN ASSIGNMENT

	Pin	Pin	
D4	1	64	D5
D3	2	63	D6
D2	3	62	D7
D1	4	61	D8
D0	5	60	D9
$\overline{AS}$	6	59	D10
$\overline{UDS}$	7	58	D11
$\overline{LDS}$	8	57	D12
R/$\overline{W}$	9	56	D13
$\overline{DTACK}$	10	55	D14
$\overline{BG}$	11	54	D15
$\overline{BGACK}$	12	53	GND
$\overline{BR}$	13	52	A23
V_{CC}	14	51	A22
CLK	15	50	A21
GND	16	49	V_{CC}
$\overline{HALT}$	17	48	A20
$\overline{RESET}$	18	47	A19
$\overline{VMA}$	19	46	A18
E	20	45	A17
$\overline{VPA}$	21	44	A16
$\overline{BERR}$	22	43	A15
$\overline{IPL2}$	23	42	A14
$\overline{IPL1}$	24	41	A13
$\overline{IPL0}$	25	40	A12
FC2	26	39	A11
FC1	27	38	A10
FC0	28	37	A9
A1	29	36	A8
A2	30	35	A7
A3	31	34	A6
A4	32	33	A5

©MOTOROLA INC., 1982   ADI-814-R3

INSTRUCTION SET

Mnemonic	Description	Operation	X	N	Z	V	C
ABCD	Add Decimal with Extend	$(Destination)_{10}$ + $(Source)_{10}$ → Destination	•	U	•	U	•
ADD	Add Binary	(Destination) + (Source) → Destination	•	•	•	•	•
ADDA	Add Address	(Destination) + (Source) → Destination	—	—	—	—	—
ADDI	Add Immediate	(Destination) + Immediate Data → Destination	•	•	•	•	•
ADDQ	Add Quick	(Destination) + Immediate Data → Destination	•	•	•	•	•
ADDX	Add Extended	(Destination) + (Source) + X → Destination	•	•	•	•	•
AND	AND Logical	(Destination) Λ (Source) → Destination	—	•	•	0	0
ANDI	AND Immediate	(Destination) Λ Immediate Data → Destination	—	•	•	0	0
ASL, ASR	Arithmetic Shift	(Destination) Shifted by <count> → Destination	•	•	•	•	•
BCC	Branch Conditionally	If CC then PC + d → PC	—	—	—	—	—
BCHG	Test a Bit and Change	~(<bit number>) OF Destination → Z ~(<bit number>) OF Destination → <bit number> OF Destination	—	—	•	—	—
BCLR	Test a Bit and Clear	~(<bit number>) OF Destination → Z 0 → <bit number> OF Destination	—	—	•	—	—
BRA	Branch Always	PC + d → PC	—	—	—	—	—
BSET	Test a Bit and Set	~(<bit number>) OF Destination → Z 1 → <bit number> OF Destination	—	—	•	—	—
BSR	Branch to Subroutine	PC → SP@ —; PC + d → PC	—	—	—	—	—
BTST	Test a Bit	~(<bit number>) OF Destination → Z	—	—	•	—	—

Condition Codes

Instruction		Operation	X	N	Z	V	C
CHK	Check Register against Bounds	If Dn <0 or Dn > (<ea>) then TRAP	—	•	U	U	U
CLR	Clear an Operand	0 → Destination	—	0	1	0	0
CMP	Compare	(Destination) − (Source)	—	•	•	•	•
CMPA	Compare Address	(Destination) − (Source)	—	•	•	•	•
CMPI	Compare Immediate	(Destination) − Immediate Data	—	•	•	•	•
CMPM	Compare Memory	(Destination) − (Source)	—	•	•	•	•
DBcc	Test Condition, Decrement and Branch	If ~CC then Dn − 1 → Dn; if Dn ≠ − 1 then PC + d → PC	—	—	—	—	—
DIVS	Signed Divide	(Destination)/(Source) → Destination	—	•	•	•	0
DIVU	Unsigned Divide	(Destination)/(Source) → Destination	—	•	•	•	0
EOR	Exclusive OR Logical	(Destination) ⊕ (Source) → Destination	—	•	•	0	0
EORI	Exclusive OR Immediate	(Destination) ⊕ Immediate Data → Destination	—	•	•	0	0
EXG	Exchange Register	Rx ↔ Ry	—	—	—	—	—
EXT	Sign Extend	(Destination) Sign-extended → Destination	—	•	•	0	0
JMP	Jump	Destination → PC	—	—	—	—	—
JSR	Jump to Subroutine	PC → SP@−; Destination → PC	—	—	—	—	—
LEA	Load Effective Address	Destination → An	—	—	—	—	—
LINK	Link and Allocate	An → SP@−; SP → An; SP + d → SP	—	—	—	—	—
LSL, LSR	Logical Shift	(Destination) Shifted by <count> → Destination	•	•	•	0	•
MOVE	Move Data from Source to Destination	(Source) → Destination	—	•	•	0	0
MOVE to CCR	Move to Condition Code	(Source) → CCR	•	•	•	•	•
MOVE to SR	Move to the Status Register	(Source) → SR	•	•	•	•	•

• affected   0 cleared   U defined
— unaffected   1 set

173

INSTRUCTION SET (CONTINUED)

Mnemonic	Description	Operation	X	N	Z	V	C
MOVE from SR	Move from the Status Register	SR → Destination	—	—	—	—	—
MOVE USP	Move User Stack Pointer	USP → An; An → USP	—	—	—	—	—
MOVEA	Move Address	(Source) → Destination	—	—	—	—	—
MOVEM	Move Multiple Registers	Registers → Destination (Source) → Registers	—	—	—	—	—
MOVEP	Move Peripheral Data	(Source) → Destination	—	—	—	—	—
MOVEQ	Move Quick	Immediate Data → Destination	—	•	•	0	0
MULS	Signed Multiply	(Destination)*(Source) → Destination	—	•	•	0	0
MULU	Unsigned Multiply	(Destination)*(Source) → Destination	—	•	•	0	0
NBCD	Negate Decimal with Extend	0 – (Destination)₁₀ – X → Destination	•	•	U	•	•
NEG	Negate	0 – (Destination) → Destination	•	•	•	•	•
NEGX	Negate with Extend	0 – (Destination) – X → Destination	•	•	U	•	•
NOP	No Operation	—	—	—	—	—	—
NOT	Logical Complement	~(Destination) → Destination	—	•	•	0	0
OR	Inclusive OR Logical	(Destination) v (Source) → Destination	—	•	•	0	0
ORI	Inclusive OR Immediate	(Destination) v Immediate Data → Destination	—	•	•	0	0
PEA	Push Effective Address	Destination → SP@–	—	—	—	—	—
RESET	Reset External Devices	—	—	—	—	—	—
ROL, ROR	Rotate (Without Extend)	(Destination) Rotated by <count> → Destination	—	•	•	0	•

Condition Codes

Mnemonic	Description	Operation	X	N	Z	V	C
ROXL, ROXR	Rotate with Extend	(Destination) Rotated by <count> → Destination	•	•	•	0	•
RTE	Return from Exception	SP@+ → SR; SP@+ → PC	•	•	•	•	•
RTR	Return and Restore Condition Codes	SP@+ → CC; SP@+ → PC	•	•	•	•	•
RTS	Return from Subroutine	SP@+ → PC	–	–	–	–	–
SBCD	Subtract Decimal with Extend	(Destination)10 – (Source)10 – X → Destination	•	U	•	U	•
SCC	Set According to Condition	If CC then 1's → Destination else 0's → Destination	–	–	–	–	–
STOP	Load Status Register and Stop	Immediate Data → SR; STOP	•	•	•	•	•
SUB	Subtract Binary	(Destination) – (Source) → Destination	•	•	•	•	•
SUBA	Subtract Address	(Destination) – (Source) → Destination	–	–	–	–	–
SUBI	Subtract Immediate	(Destination) – Immediate Data → Destination	•	•	•	•	•
SUBO	Subtract Quick	(Destination) – Immediate Data → Destination	•	•	•	•	•
SUBX	Subtract with Extend	(Destination) – (Source) – X → Destination	•	•	•	•	•
SWAP	Swap Register Halves	Register [31:16] ↔ Register [15:0]	–	•	•	0	0
TAS	Test and Set an Operand	(Destination) Tested → CC; 1 → [7] OF Destination	–	•	•	0	0
TRAP	Trap	PC → SSP@–; SR → SSP@–; (Vector) → PC	–	–	–	–	–
TRAPV	Trap on Overflow	If V then TRAP	–	–	–	–	–
TST	Test an Operand	(Destination) Tested → CC	–	•	•	0	0
UNLK	Unlink	An → SP; SP@+ → An	–	–	–	–	–

[ ] = bit number

- affected   0 cleared   U defined
- unaffected   1 set

175

Glossary

address—A specific byte in the computer's memory.

addressing modes—Refers to the form an instruction must take in order to access or handle data in memory or within the microprocessor's registers.

ASCII codes—A standard code representing numbers, letters, and symbols (such as #, $, %, *, and the like established so that one brand of computer could "talk" or communicate with another. ASCII stands for the American Standard Code for Information Interchange.

ASM16—A program that creates an intermediate file from a source code created by an editor program. This file is then placed in a final machine readable form by using the LINK16 program.

assembly language—The actual computer circuitry only understands instructions in the form of ones and zeros, called base two or binary. Assembly language is a low-level language which allows us to use symbols and letters that are easier for us humans to work with. Once we have created a program with an editor, the assembler program interprets the commands we type in and converts them to the binary codes needed by the computer.

BASIC—An acronym for Beginner's All-purpose Symbolic Instruction Code. This is the most popular language of microcomputers. BASIC is easier to write programs in than assembly language, but it does not execute them as fast.

binary—Base two. The binary number system is used by computer logic circuits. Only two states exist, represented by a one or a zero. All computer programs must be placed in binary codes before the computer can execute them. This is true regardless of the high-level language in which the program is written.

bit—A binary digit. Eight bits comprise one byte. There is no smaller accessible entity by programmers

branch—Usually the computer executes each instruction in a program in the order in which it is placed in memory. Once a starting address is defined the computer works in ascending address order. Certain instructions cause the machine to jump or "branch" out of this normal pattern and go immediately to another address located elsewhere in the program.

buffer—In programming, a buffer refers to an area or block of memory where information can be stored. Many times this block is used as a temporary storage area where data can be held as it passes from one place to another. For example, a buffer area may be used as a place where information can go on its way to and from a disk file.

bus—An electronic path over which data is transmitted.

byte—A unit of measure of a computer's memory. It takes one byte to store one alphanumeric character. Eight bits make up one byte.

chip—A common nickname for an integrated circuit or "IC." The many thousands of electronic components which comprise the integrated circuit are built onto a small piece or "chip" of material, typically silicon.

conditional branch—A program instruction where normal program flow may or may not jump to another location and begin executing instructions there. Whether or not the branch is made depends on whether or not certain "conditions" are met.

counter—A register or storage area where a value is placed and either incremented or decremented depending on whether a count up or count down is desired.

CPU—Abbreviation for central processing unit. In desktop computers the CPU is the microprocessor.

debugging—Locating and correcting syntax and logical mistakes or errors in a program. Debugging refers to getting the "bugs" out of a program.

decrement—Refers to subtracting a number, usually a one, from a

number, that is, decreasing a number by a specific value. In BASIC the equivalent would be $X=X-1$. The opposite operation is increment.

directive—Another term for pseudo-op (see "PSEUDO OP").

disk access—The process of communicating with a disk file, and transferring information to and from it.

disk file—A file is a group of records, each being similar in format. Analogous to an index file found in the home or office.

disk operating system—A series of machine-language programs which contain many routines that allow the microprocessor and associated circuitry to communicate with various external devices (peripherals) such as the video display, disk drives, modems, printers, and the keyboard.

diskette (or disk)—A flat circular plastic sheet with one or both sides covered with a ferromagnetic material. These mass storage devices are also known as "floppy diskettes." However, the term diskette has gained little acceptance, and so the term disk is generally used and preferred.

double sided—Refers to floppy disk storage. If a disk is covered with the ferromagnetic material on both sides, information can be stored on both sides. Double-sided disks therefore can store twice as much information as "single sided" disks.

EDIT16—A program which allows us to create and modify a source code for the MC68000 microprocessor on the Model 16. This code can then be put in a machine-readable form by use of an assembler.

file—See "DISK FILE."

file name—Many programs and files can be stored on a disk. To identify each one, the computer allows a file name. Basically, a file name can be up to eight letters in length. Specific rules exist for the placement of characters in the file name other than alphabetical letters.

flag—Each bit in the status register is called a flag bit. A bit will have a value of a one or zero to indicate that either something has or has not taken place. Typically, after two numbers are added together with an ADD instruction, one specific bit referred to as the "carry bit" will be either set or cleared (one or zero) depending upon whether or not the results of the addition produced a carry digit.

floppy diskette (or disk)—See "DISKETTE."

hexadecimal—Base sixteen.

IC—See "INTEGRATED CIRCUIT."

increment—Refers to adding a number to another number, that is, increasing it by a certain value. The opposite operation is decrement. In BASIC a variable is incremented with the instruction X = X + 1.

integrated circuit—An electronic component made up of thousands of individual components, all connected together in one package. (Also see "CHIP".)

LINK16—A program which takes a file created by the EDIT16 and assembled by the Assembler-16 programs and creates a final machine-executable program.

machine language—A binary coded set of instructions which the microprocessor can utilize. All programs, regardless of the high level in which they were written, must be placed in machine language as the final machine-executable form.

memory—Electronic circuits in the computer which hold binary values.

menu—When the video display shows a list of options or selections which the operator can choose from, it is referred to as a "menu." By selecting the option of his cohice, the user can branch to sections of the program which will carry out various jobs.

microprocessor—This is an electronic device called an integrated circuit, commonly known as a "chip." The microprocesor is the brains of the computer. It fetches each instruction from a program which is stored in memory and carries out a specific operation.

mnemonic—Op codes are sometimes referred to as mnemonics since they are usually comprised of several letters which represent a particular instruction. Often the mnemonic jogs one's memory to help him remember the function of the code, e.g., LD for "load," MOV for "move," and the like.

object code—The final form of a program which was created by an assembler (from a source code). The machine can execute this code directly.

peripheral—A peripheral is a device which connects to the computer and its associated circuitry. Printers, disk drives, and

modems are examples of peripheral units.

program—A program is a set of instructions that causes a computer to perform a certain task. There are many different languages in which we can write programs to make the computer do work for us. Each language has its advantages and disadvantages.

pseudo op—These are instructions which the microprocessor does not understand. The assembler translates these commands into machine codes which the microprocessor can use. Such instructions make writing assembly language programs much easier for the programmer. The Radio Shack Assembler-16 manual refers to pseudo ops as "directives."

read/write—Refers to getting information out of and placing data into a disk file.

record—A record is a piece of a file. All of the data which pertains to one case comprises a record. Many records together make up a file.

register—A small piece of RAM within the microprocessor (the central processing unit). Information can be stored in the registers and acted upon. Instructions for the microprocessor are placed in the registers.

relative address—The Assembler-16 shows us the relative address of each instruction in a program. The first instruction is given an address of 00000000 hexadecimal. This is not the actual memory location where the program will reside. By giving it a relative address, the computer can find a home for a program by locating it at the first available memory space after the disk operating system and any other programs have been loaded.

single-sided—Refers to storage on floppy disks. If only one side of a disk is coated with the proper ferromagnetic material then it is called a "single-sided" disk. Disks that can store information on both sides are known as "double-sided."

source code—Assembler programs allow us to write instructions using mnemonics and symbols. This is called a source code. It must be assembled into an object code which the machine can execute.

supervisor routines—The disk operating system has many subroutines built into it which instruct the computer how to perform operations with peripheral devices, the keyboard, video display, modems, printers, and disk drives.

References

Barden, William Jr. *TRS-80 Assembly Language Programming.* Catalog No. 62-2006 (Radio Shack).

Bartee, Thomas C. *Digital Computer Fundamentals.* New York, NY: McGraw-Hill Book Co., 1980.

Gibson, Glenn, and Liu, Yu-Cheng. *Microcomputers for Engineers and Scientists.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1980.

Kane, Gerry. *The 68000 Microprocessor Handbook.* Berkeley, CA: Osborne/McGraw-Hill, Inc., 1981.

Leventhal, Lance A. *Z80 Assembly-Language Programming.* Berkeley, CA: Osborne/McGraw-Hill, Inc., 1980.

Scanlon, Leo J. *The 68000: Principles and Programming.* Indianapolis, IN: Howard W. Sams & Co., Inc., 1981.

Radio Shack. Radio Shack's Editor Assembler Manual for the Model 16. Catalog No. 26-6040.

Index